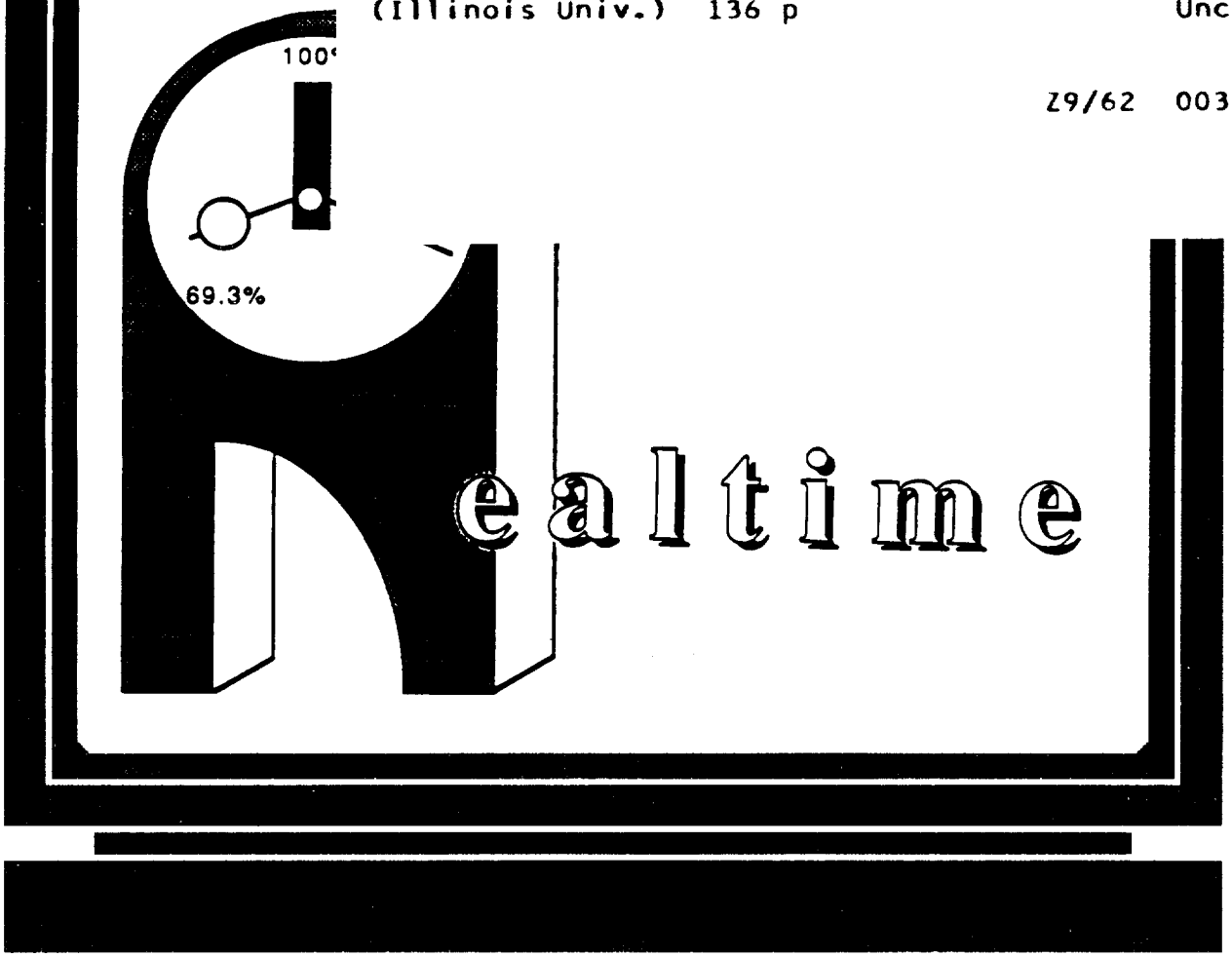
DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN



Report No. UIUCDCS-R-92-1738

UILU-ENG-92-1718

Scheduling Parallel Real-Time Tasks That Allow Imprecise Results

by

Albert C. Yu

March 1992

Report No. UIUCDCS-R-92-1738

SCHEDULING PARALLEL REAL-TIME TASKS THAT ALLOW IMPRECISE RESULTS

BY

ALBERT CHUANG-SHI YU

B.A., University of California, 1985
M.S., University of Illinois at Urbana-Champaign, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

SCHEDULING PARALLEL REAL–TIME TASKS THAT ALLOW IMPRECISE RESULTS

Albert Chuang–Shi Yu, Ph.D.
Department of Computer Science
University of Illinois at Urbana–Champaign, 1992
Kwei–Jay Lin, Advisor

Imprecise computation and parallel processing are two techniques for avoiding timing faults and tolerating hardware faults in hard real–time systems. When a result of the desired quality cannot be produced in time, hard real–time systems can produce an intermediate result of acceptable quality by imprecise computation, reduce the response time of the result by parallel processing, or both, to avoid timing faults. To mask hardware faults, a real–time task is replicated into several copies which are executed on distinct processing elements. The imprecise computation technique provides hard real–time systems with flexible functionality by trading off the quality of the result produced by a task with the amount of the computational resources required to produce it and thus enables the systems to reduce their computational loads in case of hardware faults. These two techniques permit the performance of hard real–time systems to remain predictable and to degrade gracefully.

This thesis describes efficient algorithms for scheduling two different task models on multiprocessors. Both task models support the imprecise computation technique whereby each task is logically decomposed into a hard task and a soft task. The hard task must be completed before the deadline to produce an acceptable result. The soft task refines the result produced by the hard task until the deadline. In the parallelizable task model, each task may be decomposed into concurrent subtasks which are processed simultaneously by multiple processing elements. The overhead associated with concurrent processing is assumed to be a linear function of the degree of parallelism. The scheduling algorithm for this model is optimal, if the multiprocessing overhead is indeed a linear function of the degree of parallelism. In the replicated task model,

the replicas of a task must be assigned to distinct processing elements. The performance of the scheduling algorithms for this model is evaluated by stochastic analysis and computer simulations.

To my parents,

Oren A–Jen Yu and Sue–Mei Chen,

for their love, support, and encouragement.

Acknowledgement

I would like to express my utmost gratitude to my Ph.D. thesis advisor Prof. Kwei–Jay Lin. He introduced me to and taught me about real–time systems. He was very patient with my research ideas and took time to discuss the problems with me, even when my demands were untimely. His guidance, wisdom, and humor were beneficial to me. The kindness and generousity of his family also made my stay in Illinois a joy.

Thanks to the members of my committee: Professors Jane Liu, Geneva Belford, Pravin Vaidya, and Tony Ng for their valuable comments. I learned the state of the art real–time systems from the real–time systems research group under Jane's direction. Her insights on real–time systems gave the answers to my many puzzling questions, while her enthusiasm about the problems oftentimes excited me so much that I wanted to solve them. Prof. Geneva Belford was very helpful. Our discussions were profitable to me.

I would like to thank Prof. Lin's students. Through the exchange of research ideas and important publications, they enriched my knowledge. Their friendship made my stay in Illinois seemed shorter. Ching–Chih Han, in particular, gave me helpful advice on the research of real–time systems. Susan Vrbsky kindly offered to share her terminal and lovely decorated office with me. Graig Stunkel's excellent implementation of the simplex algorithms is appreciated. Thanks to Commodore Bob Janssens and the members of the Illini Sailing Club for sharing the art and joy of sailing in central Illinois.

The love and support of my family made my pursuit of high academic endeavor possible. I am very fortunate and grateful for my parents' encouragement and unconditional support. My wife, Kyung Sun, gave me the needed love, understanding, and patience. Her artistic perspectives oftentimes cast a colorful light and enriched my thought processes. My brother, Robert, was very helpful in time of need.

It is impossible for me to complete this thesis without the guidance and support of Dr. Henry Lum and Dr. Jerry Yan in the Intelligent Systems Technology Branch at the NASA Ames Research Center. They helped me apply for the NASA Graduate Student Researchers Program and supported me generously during these years.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

Real–time embedded systems are computer control systems that require computations with timing constraints to ensure their correctness and integrity [6, 35, 39, 79, 94]. The application of real–time systems is very broad, including computer video games, automobiles, on–line medical databases, monitoring systems for intelligent highway and vehicle systems, life supporting systems, nuclear power plants, assembly line processing, avionic coordination, and robotic automation. In *hard* real–time systems, it is critical to complete the execution of tasks no later than the specified finish times. Otherwise, timing faults are said to occur. The results produced by the late tasks are of little or no use and the consequence of missing deadlines could be catastrophic and monetary loss. As a result, the integrity and correctness of hard real–time systems depends not only on the functional (logical) results of the computations but also on the time at which the results are produced.

Recently, there is a common consensus among the researchers and designers of hard real–time systems that the most important feature or property of hard real–time systems is *predictability* [84]. A predictable real–time system always produces an acceptable result on time for any given set of inputs, resources, and environments. In other words, its functional and timing behavior should be as deterministic as necessary to satisfy the system specification.

Predictable real–time systems, by definition, are fault–tolerant. If hard real–time systems cannot tolerate certain hardware or software faults, the computations of critical tasks may be interrupted and delayed, resulting in timing faults. Furthermore, (transient) hardware faults can lead to the (temporary) loss of computational resources which may result in system service interruption and unpredictable behavior.

1.1. System Issues

Four important system issues need to be addressed in the construction of predictable real–time systems.

Timing properties. To make the timing properties of real–time programs predictable, tools have been developed for the worst–case timing analysis of real–time programs [29,76,85]. The worst–case execution time of a program is the longest path in its dynamic program call graph. It is not easy to determine the longest path of a program because the length of a path can be data dependent and architecture dependent. For example, an if statement has a true branch and a false branch; which branch to take at run time depends on the data values assigned to the variables in its predicate expression. Similarly, the number of iterations in a loop statement is determined by the value of its index variable. Embedded in computer architecture are resource sharing protocols or algorithms. They decide how to share the limited resources such as caches and communication bandwidth among the programs in a computer system. The decisions affect the execution speed of each program. Furthermore, to bound the execution times of real–time programs, the loop statements and recursive functions that do not have a finite number of iterations and recursions, respectively, are banned from real–time programming. Consequently, real–time programming becomes unnatural and is a formidable task.

Efficiency. Although the worst–case timing analysis may permit some variations in task parameters, the worse–case estimation often results in an under–utilization of system resources. When valuable computing resources are wasted, the systems become more costly to build. Hence, it is important to design flexible and efficient real–time systems. Also important is the design of efficient scheduling algorithms which keep the scheduling and execution overheads small, especially in a parallel processing environment. When the overheads are small, real–time programs are executed efficiently and are more likely to meet their timing constraints.

Dynamic situations. Depending on the operation mode, a real–time system may have to satisfy different sets of timing constraints at different times. It is important to design schedulers

to make resource allocation decisions for different operation modes. The design effort should center on, in particular, those schedulers that can reconfigure systems to accommodate changing requirements so as to create minimal disruption to the ongoing operation. Moreover, some real–time systems are designed to operate in a hostile environment or for a long period of time without repairs. Examples include space shuttles, interplanetary space probes, and deep–sea submarines. The computer hardware in these systems may wear out with time. Program bugs may be uncovered in the software. The systems may have transient faults. These systems must use the run–time information from the computing environment to adjust themselves dynamically to less computational resources in a graceful and timely fashion.

Maintenance. Many real–time systems are built to last for a long period of time and are expected to evolve, when new and better technology is available. Though future modifications are expected, the exact details of the modifications may not be known completely at the design time. Consequently, these systems must be designed for maintainability.

1.2. Flexible Real–Time Systems

One approach to addressing these system issues is to develop flexible and efficient real–time systems. A flexible real–time system uses different algorithms or different hardware configurations to produce predictable results, when the timing conditions or environment vary. When it is difficult to schedule all tasks to meet their timing constraints, a flexible real–time system will leave less critical tasks unfinished to avoid timing faults. A flexible real–time system always ensures an acceptable response in the face of variations in resource availability. If a system can guarantee an acceptable result under all operating conditions, it is by definition predictable.

We use two general techniques to build flexible real–time systems: (1) the *imprecise computation* technique and (2) parallel processing. These two techniques are not mutually exclusive of each other and thus can be applied jointly.

Some real–time tasks have dynamic algorithms with stochastic execution times. Computer components may suffer from transient or permanent failures. In such cases, real–time systems may be overloaded and some tasks may not have enough time to complete their executions before the deadlines. In real–time applications such as digitized voice transmission, radar tracking, and seismic exploration, timely approximate results may be preferred over late exact results. In the transmission of digitized picture or voice, one may prefer to have intelligible voices or fuzzy pictures in time than clear voices or perfect pictures late. When radar detection and tracking systems cannot calculate the accurate coordinates before the deadline, one may like to have the proximities. The geological models used in seismic exploration are imprecise, and precise models do not exist. One may prefer to have an approximate model in time than a more precise model late. What these real–time applications allude to is that when a real–time system is overloaded, a task, although assigned with less than the complete processing time, may still produce an *acceptable* result.

We use the imprecise computation technique to ensure that an approximate result of an acceptable quality is available, whenever the exact result of the desired quality cannot be obtained in time [56, 56, 63]. This technique prevents timing faults, achieves graceful degradation, and thus makes real–time systems more flexible. Since the advent of the imprecise computation technique, several workload models have been developed [58]. These models trade off the quality (specificity or precision) of the result produced by a real–time task with the amount of computation time needed to produce it. A task in these models can be logically decomposed into a hard task and a soft task. The hard task must be completed before the deadline to produce an acceptable result. The soft task is executed after the hard task in order to improve the quality of the result until the deadline. The concept of imprecise computation has been applied in a number of related fields including iterative algorithms (heuristic search, numerical algorithms, the progressive buildup method for facsimile transmission [93], and the successive doubling method for computing fast Fourier transformations), multi–phased algorithms, statistical or

Monte Carlo methods, and approximate relational algebra or object–oriented database query processing [12, 81, 90].

The other technique is parallel processing. Recent advances in parallel processing technology improve the execution speed of real–time computations and increase the likelihood that real–time applications will meet their timing requirements. To avoid timing faults, real–time system designers can reduce the execution times of real–time tasks by utilizing efficient parallel algorithms designed for a variety of powerful and cost–effective multiprocessors [5, 32]. As a result, the design of real–time systems is more flexible because system designers can choose from a number of parallel algorithms and multiprocessors.

In this research, we focus on the design of efficient multiprocessor scheduling algorithms for imprecise computations. The goal is two–fold: (1) to find the optimal schedule, if it exists, and (2) to reduce the scheduling and concurrent execution overheads. A schedule is optimal when the weighted sum of all unexecuted portions of the soft tasks is minimized. If the scheduling algorithms are efficient and keep the scheduling and concurrent execution overheads small, then the multiprocessor systems have good performance.

1.3. Problems Associated with Multiprocessor Scheduling of Imprecise Computations

We identify several difficulties associated with scheduling imprecise computations on multiprocessors.

Complexity. Although imprecise computation and parallel processing add flexibility in the design of real–time systems, the flexibility does not reduce the complexity of the real–time multiprocessor scheduling problem. For example, some real–time tasks are not parallelizable and can only be executed sequentially. The imprecise computation technique may not be applicable to other real–time tasks. The real–time multiprocessor scheduling problem for these tasks only has been shown to be NP–complete [25]. Therefore, the more general problem of scheduling

parallelizable imprecise computations on a multiprocessor system is at least as hard as an NP–complete problem.

Multiprocessing overheads. Previous research in optimal real–time scheduling algorithms assumes negligible multiprocessing or multiprogramming overheads and thus simplifies the problem. However, multiprocessing overheads (including process spawning, process preemption and migration, data distribution, memory interference, cache coherence, and interconnection network contention and blocking) incur significant additional costs and cannot be ignored.

Scheduling constraints. Fault–tolerance techniques may impose additional constraints on tasks and make the scheduling problems more difficult. For example, when the n–modular redundancy technique is used, the clones or copies of a task must be allocated to distinct processors [7]. Furthermore, hardware failures occur dynamically and may reduce the amount of computational resources. It is important to design efficient scheduling algorithms that can either modify the old schedule with a minimal change or construct a new schedule quickly so that real–time systems can operate continuously and remain predictable.

1.4. Related Research

In this section, we give a general introduction to the research relevant to the scheduling of imprecise computations. Each chapter of the thesis presents the research closely related to the problem addressed in the chapter. Liu et al. gave an overview of the problems in scheduling imprecise computations and described the algorithms for solving these problems [58]. Shih et al. designed an optimal algorithm for scheduling aperiodic imprecise computations on a single processor and multiprocessors [78]. Chung et al. designed several heuristic algorithms for scheduling periodic imprecise computations on multiprocessors [14]. Leung et al. presented scheduling algorithms for minimizing the mean flow time of imprecise computations with an error constraint [52]. The error constraint is that the total unfinished portion of the soft tasks is no more than a given threshold. They also gave algorithms for maximizing the number of on–time imprecise

computations with the error constraint [53]. Mok and Dertouzos showed that optimal scheduling without a priori knowledge of the ready times of the tasks is impossible except for the case of a uniprocessor system [62]. Leung and Whitehead studied the complexity of the problem: determining whether a set of periodic tasks can be scheduled on identical multiprocessor systems with respect to static priority–driven scheduling algorithms. They showed that this problem is NP–hard [51]. Rajkumar developed a resource access control protocol based on the priority ceiling protocol for shared memory multiprocessors [67]. Stankovic et al. developed algorithms for scheduling tasks with communication overheads, time, and resource constraints in a distributed environment [83, 97, 98]. There are a number of excellent books and survey papers on task scheduling [11, 15, 22, 44, 49, 50]. However, the multiprocessing overhead problem and the task reconfiguration problem raised in the previous section remain unsolved.

1.5. Organization of this Thesis

The problem of scheduling imprecise computations on a multiprocessor system is complex. The solution may be specific to the assumptions made in the computation model. We use the following general approach to address the scheduling problem. First, we extend the imprecise computation model with additional constraints and assumptions. The idea is similar to the taxonomy schemes used for showing the relations among the classic scheduling problems [11, 44, 50]. Second, we develop multiprocessor scheduling algorithms for each extended computation model.

The remainder of this thesis is organized into five chapters. In Chapter 2, we present the base model upon which different models are extended in the other chapters. We describe the traditional task system and augment it with the imprecise computation model. This chapter also gives the processor–level multiprocessor model used for our study.

In Chapter 3, we assume that each imprecise computation may be parallelized and executed on multiple processors with a multiprocessing overhead which is a linear function of the degree of parallelism. We show that the problem of time allocation in such a real–time application can be

formulated and solved as a linear programming problem. We also present an algorithm for constructing a multiprocessor schedule from the linear programming solution. This algorithm guarantees that the multiprocessing overhead generated in the multiprocessor schedule does not exceed a linear upper bound.

In Chapter 4, we present two algorithms for scheduling periodic tasks on a multiprocessor system under a fault–tolerant requirement. Our approach incorporates both the redundancy and masking technique, and the imprecise computation model. Since the tasks in hard real–time systems have stringent timing constraints, the redundancy and masking technique is more appropriate than the rollback techniques which usually require extra time for error recovery. The imprecise computation model provides flexible functionality and therefore permits the performance of a real–time system to degrade gracefully. We evaluate the algorithms by stochastic analysis and Monte Carlo simulations. The results show that the algorithms are resilient under hardware failures.

Chapter 5 presents the system supports for fault detection and removal which the operating system is assumed to have in Chapter 4. We describe several schemes for uncovering different classes of faults and a recovery manager which uses the algorithms given in Chapter 4 to reconfigure tasks in the system, when computer components fail.

Chapter 6 contains the conclusions and suggestions for future work.

Chapter 2
The Base Model

This chapter presents the common components of the scheduling problems addressed in this thesis. Section 2.1 describes the traditional task system. In Section 2.2, we define the monotone imprecise computation model and incorporate the model in the traditional task system. Section 2.3 shows the equivalence of two objective functions. Section 2.4 presents the multiprocessor model.

2.1. The Traditional Task System

A task system, $TS = \{T^1, T^2, \ldots, T^n\}$, for a real-time application is a directed acyclic graph in which the nodes represents aperiodic tasks, and the arcs represents dependence and AND/OR relations [11, 15, 22, 44, 50]. Henceforth, we use the right superscript of a variable to denote the task ID in this thesis. Associated with each task T^j is a weight w^j which indicates the relative importance of T^j.

Each aperiodic task T^j in TS is characterized by the following rational parameters:

(1) ready time r^j, the time instance at which T^j is ready for execution,

(2) deadline d^j, the time instance by which T^j must be terminated, and

(3) complete processing time τ^j, the total time required to execute T^j if T^j is executed sequentially.

If TS has periodic tasks, we can use the following transformation technique to describe periodic tasks in terms of aperiodic tasks. The periodic task graph shows the dependence and AND/OR relations among the periodic tasks in the minor and major cycles. A periodic task T^j is characterized by the following rational parameters:

(1) initial request time i^j, the first time instance at which T^j is requested for execution,

(2) period p^j, the duration between any two consecutive requests for executing T^j, and

(3) complete processing time τ^j, the duration required to service the request of T^j if T^j is executed sequentially.

The k^{th} request for executing T^j is initiated $i^j + (k-1)p^j$. The deadline for completing the k^{th} request is $i^j + kp^j$.

A TS of periodic tasks (PTS) can be transformed to an equivalent TS of aperiodic tasks (ATS), but not vice versa. A periodic task can be represented by an infinite sequence of aperiodic tasks. If we are interested in a cyclic schedule for PTS, the infinite sequences of aperiodic tasks can be truncated to finite sequences. It is sufficient to keep those aperiodic tasks whose life times ($[r^j, d^j]$) are contained in the interval $[t, t + LCM]$. t is the critical instance of PTS [57]. LCM is the least common multiple of all periods in PTS. Indeed, it has been shown that by such transformation a cyclic multiprocessor schedule can be constructed for a PTS [46].

2.2. Imprecise Computation Model

In a real–time application that supports the imprecise computation model, a task T^j can be logically decomposed into two subtasks: the hard (real–time) subtask, H^j and the soft (real–time) subtask, S^j. The request or ready time and the deadline of the tasks H^j and S^j are the same as those of T^j. We define h^j and s^j to be the processing times of H^j and S^j, respectively; therefore, $h^j + s^j = \tau^j$. When $h^j = 0$, T^j is a soft task. When $s^j = 0$, T^j is a hard task. When $h^j \neq 0$, task S^j can be executed only after task H^j is completed. If a task does not produce an imprecise result (i.e, a traditional task), it is a hard task.

A task T^j is said to produce an acceptable result when its hard task H^j is completed before its deadline. In other words, the total processing time x^j assigned to task T^j is no less than the processing time (h^j) of its hard task. A soft task is executed before its deadline and after the completion of the hard task with which it has a dependence relation. It is terminated at the

deadline d^j even if it is not completed at the time.

When the total processing time x^j assigned to a soft task S^j is equal to s^j, the task T^j is said to be *precisely* completed. In this case, the error ϵ^j in the result produced by T^j (or simply the error of T^j) is zero. If x^j is less than h^j, ϵ^j is one. Otherwise, ϵ^j is a function of x^j whose function value is bounded between zero and one.

A task T^j is *monotone*, if ϵ^j is a non–increasing function of x^j. Namely, as a monotone task executes longer, the quality of its result improves. Because of the discrete nature of digital computers, ϵ^j is usually a staircase function. Figure 2.1 shows an example of a monotone task.

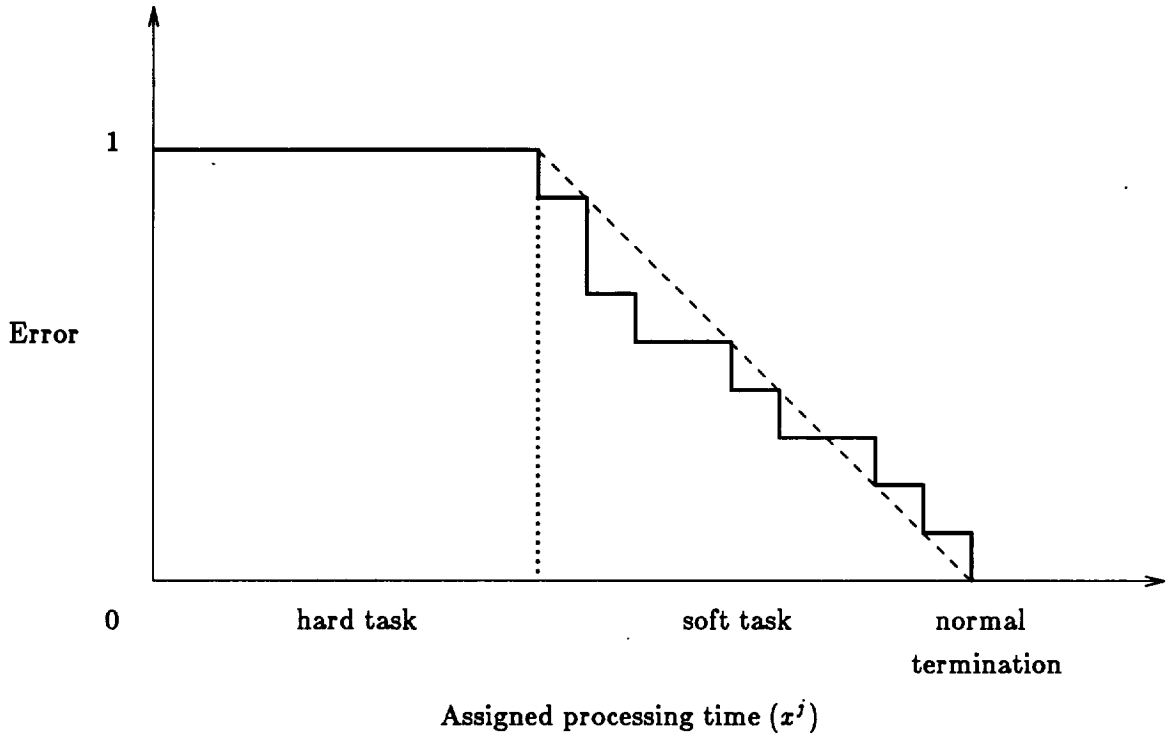


Figure 2.1: A monotone imprecise computation

Monotone tasks can be classified into two types. A task of the first type can be terminated any time during the execution of its soft task. The intermediate result produced by the task is approximate to the precise result produced if the soft task is executed to completion. Examples of the first type include heuristic search, numerical algorithms, the progressive buildup method for facsimile transmission, radar detection and tracking systems, the successive doubling method for computing fast Fourier transformations, statistical methods, and approximate database query processing.

A task of the second type has multiple phases or versions [88]. The system scheduler must determine before execution which phases to skip and which version to use for the task. If the task is terminated during its execution, the intermediate result produced is not useful. Examples of the second type include the transmission of compressed digitized picture or voice [99], image processing, and text processing in which pictures, equations, tables, or references may be omited.

The *Concord* project at the University of Illinois has developed the programming language primitives and system supports for imprecise computation [55]. The *milestone* technique developed for iterative algorithms records intermediate results periodically and returns the latest set when a deadline is reached [56, 63]. The *sieve* technique developed for multi-phased algorithms skips certain predefined sections of code.

In real-time applications that support the imprecise computation model, the task systems can be characterized by task graphs in which the nodes are imprecise computations. Figure 2.2 depicts an example of the task graphs. In this thesis, we study the task systems in which if there is a dependence relation from task T^1 to task T^2, then $d^1 \geq r^2$. Hence, the dependence relations are implied by the timing constraints and can be omitted in the description of the task systems. In addition, we approximate the error ϵ^j of T^j by $r^j - x^j$. Previous experience has shown that linear approximation is very effective. If x^j of any task T^j is less than h^j in a multiprocessor schedule, the schedule is infeasible. We assume that the error of an aperiodic task does not propagate to the next aperiodic tasks with which it has dependence relations. The error of a

periodic task does not accumulate.

2.3. Equivalence of Two Objective Functions

The objective function of the scheduling problems is to minimize the total weighted error (ϵ) of the task set TS. $\epsilon = \sum_{j=1}^{n} w^j \epsilon^j$. For PTS, it is more convenient to define the objective function as maximizing the total weighted utilization (TWU) instead of minimizing the total weighted error. TWU is $\max \sum_{j=1}^{n} w^j u^j$ where u^j is the utilization of a periodic task T^j and is defined as

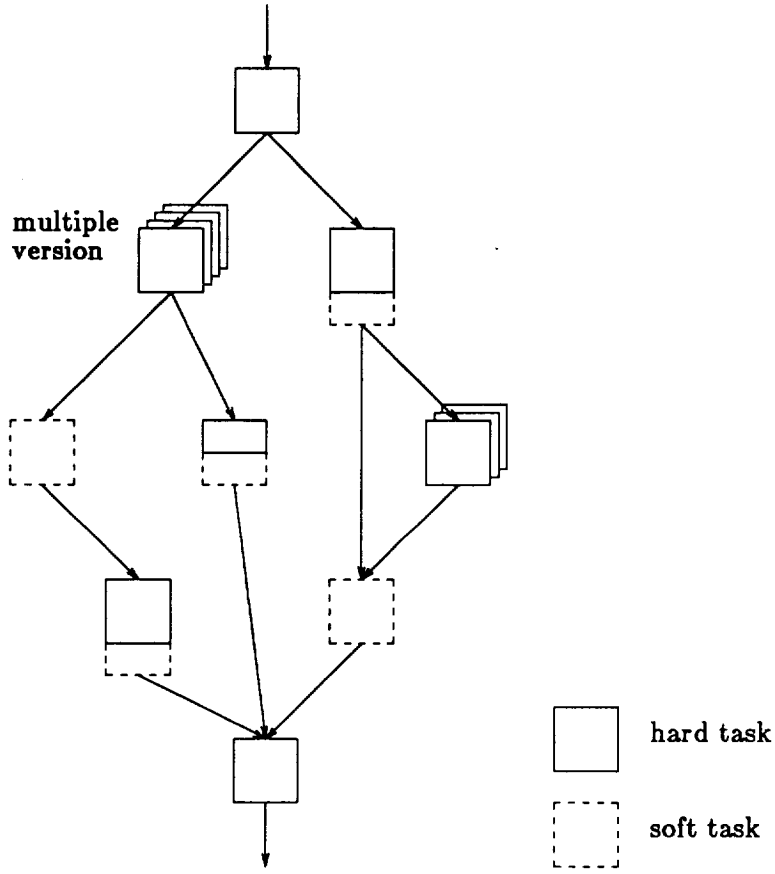


Figure 2.2: An example of the imprecise task graph

x^j/p^j. We show that $\min \sum_{j=1}^{n} w_1^j \epsilon^j$ and $\max \sum_{j=1}^{n} w_2^j u^j$ are equivalent, if we set the weight w_1^j equal to w_2^j/p^j.

$$\min \sum_{j=1}^{n} w_1^j \epsilon^j = \min \sum_{j=1}^{n} w_1^j (r^j - x^j) = \sum_{j=1}^{n} w_1^j r^j - \max \sum_{j=1}^{n} w_1^j x^j =$$

$$\sum_{j=1}^{n} w_1^j r^j - \max \sum_{j=1}^{n} (w_2^j/p^j) x^j = \sum_{j=1}^{n} w_1^j r^j - \max \sum_{j=1}^{n} w_2^j u^j$$

Since $\sum_{j=1}^{n} w_1^j r^j$ is a constant, $\sum_{j=1}^{n} w_1^j \epsilon^j$ is minimized when $\sum_{j=1}^{n} w_2^j u^j$ is maximized.

2.4. Multiprocessor Model

For our study, we assume a general multiprocessor model consisting of three main components: processors (P_i), memories, and an interconnection network. Memories are either local or global. Each local memory (LM_i) in the multiprocessor model is specific to a processor and cannot be accessed or referenced directly by the other processors. Global memories (GM_i), on the other hand, are accessible by all processors. Let processor element PE_i denote P_i and LM_i. Figure 2.3 shows the processor–level organization of the multiprocessor model.

The multiprocessor model is quite general and includes both tightly–coupled and loosely–coupled multiprocessors. For example, if the local memories of the multiprocessor model are fast caches and the interconnection network is a shared bus, then the multiprocessor model resembles Sequent or Encore multiprocessors [1, 2]. If the local memories are absent and the interconnection network is a crossbar network, then the multiprocessor model resembles C.mmp [96]. Cedar multiprocessor is similar to the multiprocessor model with a hierarchical (two–level) memories connected by an Omega interconnection network [43].

The multiprocessor model is also appropriate for loosely–coupled multiprocessors such as Cosmic Cube, Intel Hypercube, and Cm* [3, 34, 73]. For these loosely–coupled multiprocessors, the global memories are isomorphic to the local memories in the multiprocessor model. (An

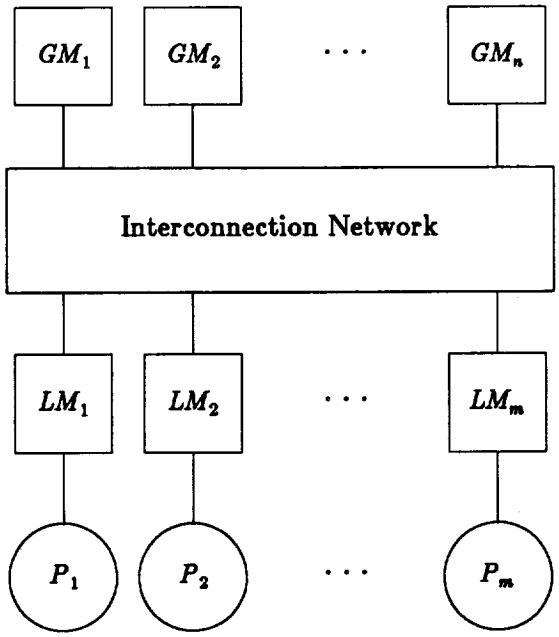


Figure 2.3: Logical organization of a multiprocessor

isomorphic function is one to one mapping with possible renaming.) The interconnection networks in Cosmic Cube and Intel Hypercube are binary interconnection networks. Cm* has a two–level interconnection network. The intracluster bus links seven or eight processor–memory pairs in a cluster. The intercluster bus provides the communication between clusters. Table 2.1 shows the classification of these multiprocessors according the memory and interconnection network components of the multiprocessor model. $GM \equiv LM$ denotes the isomorphic mapping between the global memories and the local memories. $\emptyset$ indicates the absence of local memories.

Table 2.1: Classification of some existing multiprocessors

Class	*GM*	*LM*	*Interconnection network*	*Multiprocessor*
Tightly–coupled		Cache	Bus	Sequent, and Encore
		Ø	Crossbar	C.mmp
			Omega network	Cedar
Loosely–coupled	$GM \equiv LM$		Binary network	Cosmic Cube, and Intel Hypercube
	$GM \equiv LM$		Intracluster and intercluster networks	Cm*

Chapter 3
Scheduling of Parallelizable Aperiodic Tasks

In this chapter, we describe an algorithm for solving the *imprecise multiprocessor scheduling* problem. The algorithm solves the problem in two phases. In the first phase, the problem of time allocation for parallelizable aperiodic tasks is formulated and solved as a linear programming problem. These parallelizable tasks, depending on how they are decomposed and scheduled, create different multiprocessing overheads. We characterize these multiprocessing overheads by linear functions of the degree of parallelism. Efficient algorithms designed for linear programming are used to solve the problem. If multiprocessor overheads can be accurately represented by linear models, then the time allocation is optimal in the sense that: first, each hard task is completed before its deadline, and second, the weighted sum of the unexecuted portions of the soft tasks is minimized.

The linear programming solution decides only how much time should be allocated to each task, but without specifying which of the processors should be used. In the second phase, we use an algorithm for constructing a preemptive multiprocessor schedule from the linear programming solution. The multiprocessing overhead associated with process preemptions and migrations is discrete and nonlinear, and may not be represented exactly nor bounded closely by a linear function of the degree of parallelism. While keeping the number of preemptions and migrations generated in the multiprocessor schedule small, this algorithm assures this number not to exceed a linear upper bound.

The chapter is organized as follows. Section 3.1 briefly describes the related research. In Section 3.2, we define formally the imprecise multiprocessor scheduling problem. Section 3.3 presents the linear programming formulation of the imprecise multiprocessor scheduling problem and its complexity. Section 3.4 describes the algorithm for constructing a preemptive

multiprocessor schedule from the linear programming solution. We show examples of parallelizable tasks in Section 3.5. We give the summary in Section 3.6.

3.1. Related Research

Han and Lin investigated the real–time multiprocessor scheduling problem with multiprocessing overhead functions [27]. They studied the cases where the multiprocessing overhead of a parallelizable task is either zero or a constant. Du and Leung addressed a related optimal finish time (OFT) multiprocessor scheduling problem [20]. They investigated the case where the execution time of a parallelizable task is a monotonic non–increasing function of the degree of parallelism. In particular, they showed the existence of a preemptive pseudo–polynomial time algorithm for the OFT multiprocessor scheduling problem. Our linear programming approach is different from Shih et al's network flow approach in several ways [77]. First, real–time tasks are parallelizable to different degrees of parallelism. The degree of parallelism of real–time tasks may be restricted to one. Second, multiprocessing overheads are not neglected and are included in the problem formulation. Finally, despite the fact that the imprecise multiprocessor scheduling problem is more complex, the complexity of the linear programming approach is the same as the network flow approach.

3.2. Imprecise Multiprocessor Scheduling Problem

A parallelizable aperiodic task system $PATS$ is TS composed of a collection of n parallelizable aperiodic tasks. Each task T^j (hence, H^j and S^j) in $PATS$ may be parallelized and executed concurrently on multiple processors with a multiprocessing overhead. The degree of parallelism of T^j, denoted by k^j, is the number of concurrent subtasks decomposed in T^j. The multiprocessing overhead of T^j is a function of its degree of parallelism and the execution times of its concurrent subtasks. In other words, T^j can have several versions of codes each of which has a different degree of parallelism and perhaps a different multiprocessing overhead. Let K^j denote the maximum degree of parallelism of T^j or the maximum number of concurrent tasks used in

T^j.

A multiprocessor schedule is an assignment of the (concurrent) tasks in $PATS$ to processors in disjoint intervals of time. A schedule is *feasible* when (a) every task T^j is executed after its ready time, and (b) every task T^j has an acceptable result before its deadline. A schedule is optimal (respectively, precise), if the total weighted error ϵ of the task set $PATS$ executed according to a feasible schedule is minimum (respectively, zero).

3.3. Time Allocation Problem

Given a parallelizable aperiodic task system $PATS$, we sort the ready times and deadlines of the n tasks in $PATS$ into a non-decreasing list of numbers. The duplicated entries in the list are removed and the list becomes an increasing sequence of distinct numbers, $\mathbf{a}=\{a_1, a_2, \ldots, a_l\}$ with $l \leq 2n$. In other words, $\mathbf{a}$ contains all distinct values of r_i and d_i, and a_i is either the ready time or the deadline of some task in $PATS$. The sequence $\mathbf{a}$ divides the time into l segments, $g_1, \ldots, g_l$. Segment g_i is the time interval $[a_i, a_{i+1}]$, for $i=1,2, \ldots, l-1$, while g_l is the interval $[a_l, \infty]$. We use t_i to denote the length of $[a_i, a_{i+1}]$.

A segment g_i is located within the life time $[r^j, d^j]$ of the task T^j in $PATS$, if $r^j \leq a_i < a_{i+1} \leq d^j$. Let $[a_i, a_{i+1}] \subseteq [r^j, d^j]$ denote the during (improper subset) relation. If $[a_i, a_{i+1}] \subseteq [r^j, d^j]$, we want to determine the value of the variable x_i^j which is the total processing time to be assigned to the task T^j in g_i. If $[a_i, a_{i+1}] \not\subseteq [r^j, d^j]$, x_i^j should always be 0 and is excluded from the linear programming formulation. x_i^j is nonnegative and may be greater than t_i, if concurrent processing is allowed. $\lceil x_i^j/t_i \rceil$ is the minimum number of concurrent tasks required for T^j in g^i. If x_i^j is not greater than t_i, then T^j is executed sequentially.

Suppose the multiprocessing overhead is a linear function. The imprecise multiprocessor scheduling problem can be formulated as a linear programming problem. We first define y_i^j, which is the *degree of parallel execution overhead*, as follows:

$$y_i^j = \begin{cases} 0 & x_i^j \le t_i \\ x_i^j/t_i - 1 & otherwise \end{cases}$$

y_i^j is a nonnegative number and $\lceil y_i^j \rceil$ has a value one less than the conventional definition of the degree of parallelism. When x_i^j is not greater than t_i, T^j is executed sequentially and y_i^j indicates a zero degree of overhead in g_i. Otherwise, y_i^j indicates the additional number of processors required to provide x_i^j to T^j. We let the multiprocessing overhead be a linear function $\sum_{i=1}^{l-1} o_i^j y_i^j$ without loss of generality. o_i^j represents the multiprocessing overhead for T^j in g_i. For a class of parallel algorithms with almost linear speedup, we find that their multiprocessing overheads can be bounded with such a linear function [24, 28, 30, 95]. o_i^j is a function of t_i and parallel algorithms and may be the same for all segments. More examples of parallelizable tasks can be found in Section 3.5.

3.3.1. Linear Programming (LP) Formulation

The linear programming formulation of the imprecise multiprocessor scheduling problem is defined as follows:

$$\min \sum_{j=1}^{n} w^j \epsilon^j$$

$$\epsilon^j + \sum_{i=1}^{l-1} x_i^j = r^j + \sum_{i=1}^{l-1} o_i^j y_i^j \qquad j = 1, 2, \ldots, n$$

$$\sum_{i=1}^{l-1} x_i^j \geq h^j + \sum_{i=1}^{l-1} o_i^j y_i^j \qquad j = 1, 2, \ldots, n$$

$$m t_i \geq \sum_{j=1}^{n} x_i^j \qquad i = 1, 2, \ldots, l-1$$

$$(K^j - 1) \geq y_i^j \geq 0 \qquad i = 1, 2, \ldots, l-1.\ j = 1, 2, \ldots, n$$

$$y_i^j \geq (x_i^j / t_i - 1) \qquad i = 1, 2, \ldots, l-1.\ j = 1, 2, \ldots, n$$

$$x_i^j \geq 0 \qquad i = 1, 2, \ldots, l-1.\ j = 1, 2, \ldots, n$$

The objective function in the linear programming formulation minimizes the total weighted error. The first constraint specifies that the sum of the error of a task T^j and the total processing time assigned to T^j is equal to its sequential processing time r^j and the total multiprocessing overhead. The second constraint represents that the total processing time assigned to T^j is not less than the sum of the processing time h^j of hard task H^j and the multiprocessing overhead. The first and second constraints imply that the error of T^j must be nonnegative and less than or equal to the processing time (s^j) of the soft task (S^j). The third constraint requires that the total processing time of the tasks allocated in each segment g_i is nonnegative and not greater than the total processing time available on m processors. The fourth and fifth constraints give the lower and upper bounds of the degree of overhead y_i^j for T^j. The degree of overhead y_i^j in the time interval g_i is nonnegative and bounded from above by $K^j - 1$, where K^j is the maximum number of concurrent tasks allowed for T^j. Moreover, y_i^j must be no less than $(x_i^j / t_i - 1)$.

The optimal solution for the linear programming formulation of the imprecise multiprocessor scheduling problem, if one exists, is a feasible time allocation in which the total weighted error is minimal. In fact, the allocation has the minimum weighted tradeoff between the processing time assigned and the multiprocessing overhead as demonstrated by the following equivalence transformations of the objective function:

$$\min \sum_{j=1}^{n} w^j \epsilon^j = \min \sum_{j=1}^{n} w^j (\tau^j + \sum_{i=1}^{l-1} o_i^j y_i^j - \sum_{i=1}^{l-1} x_i^j) =$$

$$\sum_{j=1}^{n} w^j \tau^j + \min \sum_{j=1}^{n} w^j \sum_{i=1}^{l-1} (o_i^j y_i^j - x_i^j)$$

The tradeoff concords with our goal in solving the imprecise multiprocessor scheduling problem. That is to utilize multiprocessing as much as possible, subject to multiprocessing overheads, to produce a feasible schedule with the minimum total weighted error. Consequently, the tradeoff between the multiprocessing overhead and the total processing time assigned are explicit in the transformed objective function.

In the case that the processing time s_i of each soft real-time task S_i is zero, the linear programming formulation is simplified nicely to a decision problem. To see this, let's examine the first and the second constraints of the linear program formulation shown below:

$$\epsilon^j + \sum_{i=1}^{l-1} x_i^j = h^j + \sum_{i=1}^{l-1} o_i^j y_i^j \qquad j = 1, 2, \ldots, n$$

$$\sum_{i=1}^{l-1} x_i^j \geq h^j + \sum_{i=1}^{l-1} o_i^j y_i^j \qquad j = 1, 2, \ldots, n$$

h^j replaces τ^j in the first equality constraint. Since ϵ^j is nonnegative, these two inequalities have feasible solutions only when $\epsilon^j = 0$. As a result, in case of $s^j = 0$ for all j's, the linear programming formulation is simplified to a decision problem. The solution is either infeasible or $\min \sum_{j=1}^{n} w^j \epsilon^j = 0$.

The solution of the linear program formulation satisfies the degree of parallel execution overhead (y_i^j) defined earlier. This can be shown by geometry interpretation. The optimal solution of a linear program always locates on a vertex of the polytope prescribed by the linear constraints [66]. In other words, the optimal point is at the intersection of the hyperplanes. Since

the objective function minimizes y_i^j and maximizes x_i^j, y_i^j is only constrained by the lower hyperplanes associated with the fourth and fifth inequalities in the formulation. Hence, y_i^j is assigned with the value as defined by the degree of parallel execution overhead.

3.3.2. Example of Time Allocation

We illustrate the linear programming formulation of the imprecise multiprocessor scheduling problem by a simple example. Given three tasks, we want to construct a two processor schedule with the minimum total weighted error. Suppose the multiprocessing overhead function is defined as $o^j \sum_{i=1}^{l-1} y_i^j$. Table 3.1 shows the parameters of each task T^j in the problem instance. Table 3.2 displays the solution of the problem instance. The a_i row gives the beginnings of the time intervals associated with x_i or y_i. A blank entry in Table 3.2 indicates that either x_i^j or y_i^j is not contained in the life time $[r^j, d^j]$ of T^j and, hence, this x_i^j or y_i^j is excluded from the linear programming formulation of the problem instance. The minimum total weighted error of the feasible schedule is 19.

A number of efficient polynomial–time algorithms for linear programming have been devised in the past [36, 37]. The complexity of the linear programming formulation of the imprecise multiprocessor scheduling problem is the same as the complexity of the most efficient algorithm for linear programming. To simplify argument, let's assume that the arithmetic operations on the parameters of the imprecise multiprocessor scheduling problem and the intermediate results can be carried out with sufficient precision in a constant number of steps. One efficient algorithm for linear programming requires $O((m+n)n^2+(m+n)^{1.5}n)$ operations where m is the number of inequalities and n is the number of variables [89].

It is possible to formulate other variations of the imprecise multiprocessor scheduling problem for linear programming. For example, replace the last inequality constraint $y_i^j \geq (x_i^j/t_i - 1)$ by $y^j \geq (x_i^j/t_i - 1)$. We have, instead of y_i^j for each time interval $[a_i, a_{i+1}]$, a maximum degree of

Table 3.1: An imprecise multiprocessor scheduling problem

	r	d	τ	h	s	w	o	K
T^1	0	6	8	3	5	3	2	2
T^2	4	12	13	10	3	2	2	2
T^3	0	14	17	7	10	1	2	2

Table 3.2: The linear programming solution for the problem instance in Table 3.1

	x_1	x_2	x_3	x_4	$\sum x_i$	y_1	y_2	y_3	y_4	$\sum y_i$	$w\epsilon$
a_i	0	4	6	12		0	4	6	12		
T^1	6	2			8	0.5	0			0.5	3
T^2		2	9		11		0	0.5		0.5	6
T^3	2	0	3	2	7	0	0	0	0	0	10

overhead y^j for each task T^j in *PATS*. The multiprocessing overhead function becomes $o^j y^j$ where o^j is a constant. The complexity of this new formulation is the same as the complexity of the original formulation.

3.4. Construction of a Multiprocessor Schedule

The linear program solution only gives us the amount of processing time allocated to the tasks in each time interval. To have a multiprocessor schedule, we still need to decide which task is to run on which processor(s) in each segment. For the problem of processor assignment, we first partition each of these tasks into concurrent subtasks which are then allocated to processors. The concurrent subtasks in each time segment must satisfy the following constraints. The processing time of a subtask is not greater than the length of the time interval. The processing times assigned to the concurrent subtasks of a task add up to its processing time. A task is not

decomposed into more concurrent subtasks than its maximum degree of parallelism. Partition and allocation of these tasks may incur process preemptions and migrations. Therefore, it is important to design an algorithm which minimizes the number of process preemptions and the number of migrations.

We have designed an algorithm for constructing a multiprocessor schedule in which the number of process preemptions and the number of migrations are guaranteed not to exceed some upper bounds. The upper bounds can be formulated as linear functions and incorporated into the linear programming formulation. The algorithm also tries to minimize the number of process preemptions and migrations. The algorithm has two phases.

3.4.1. Phase 1: Finding Task Sequences

In the first phase (Figure 3.1), the algorithm partitions each x_i^j in the linear programming solution into concurrent subtasks. It then constructs task sequences from these partitioned subtasks. To reduce the number of preemptions in a time interval $[a_i, a_{i+1}]$, the partition_subtasks procedure generates the minimum number $\lceil x_i^j/t_i \rceil$ of concurrent tasks for a task T^j. If $x_i^j \geq t_i$, then $\lfloor x_i^j/t_i \rfloor$ of these concurrent tasks are called *integral* tasks because their processing times are equal to t_i. If x_i^j is not a multiple of t_i, then there is a *fractional* task whose processing time is $f_i^j = x_i^j - \lfloor x_i^j/t_i \rfloor t_i$. Since each integral task needs a full processor and is not preempted in the time interval, the problem is simplified to that of constructing task sequences of fractional tasks on the remaining processors.

The fractional_task_seq procedure partitions the fractional tasks and allocates them to task sequences for the remaining processors. Let F_i denote an ordered set of the fractional tasks $\{f_i^j\}$ sorted in the order of decreasing processing time in the time interval $[a_i, a_{i+1}]$. To reduce the number of partitioned fractional tasks, we want to find in F_i as many task sequences as possible; each task sequence has the total processing time equal to t_i. These task sequences can be found by a pseudo–polynomial time algorithm for the sum of subset problem [33, 45].

Procedure form_sequence
input: $\{x_i^j\}$
output: I, TS $<I$ is the set of integral tasks and TS is the set of task sequences.$>$
begin
partition_subtasks($\{x_i^j\}$);
fractional_task_seq($\{f_i^j\}$);
end;

Figure 3.1: Procedure form_sequence

The fractional_task_seq procedure is a loop consisting of two steps. In the first step, it calls the algorithm for the sum of subset problem repeatedly to look for a subset of F_i whose total processing time is t_i. If found, the subset forms a task sequence and then is added to a set TS_i of task sequences. The subset is deleted from F_i. Otherwise, proceed to the second step.

In the second step, if the total processing time of F_i is less than t_i, F_i forms a task sequence and is added to TS_i. The fractional_task_seq procedure terminates. Otherwise, since F_i contains no task sequences whose total processing time is t_i, it is necessary to partition a task in F_i to construct such a task sequence. We select the first j tasks in F_i such that their total execution time is greater than t_i but the total processing time of the first $j-1$ tasks is less than t_i. These j tasks denoted by C are deleted from F_i. The j^{th} task c is partitioned to c_l and c_r where the execution time of c_l is $\sum_{f_i \in C} f_i - t_i$. The task c is deleted from the task sequence C. The task sequence $(C\ c_r)$ is added to TS_i. The task c_l is inserted to F_i, and loop back to the first step.

3.4.2. Phase 2: Sequences Matching

The procedure in the second phase (Figure 3.2) tries to reduce the number of preemptions and migrations in a multiprocessor schedule. It arranges the execution order of the tasks in a task sequence and allocates task sequences to processors so that the subtasks of the same task in

adjacent time segments are scheduled consecutively on the same processor. The problem can be formulated as a bipartite matching problem. Efficient algorithms for the bipartite matching problem can be used for reducing the number of preemptions and migrations [87]. There are, however, position constraints for fractional tasks. A fractional task in a task sequence can be at either the left–most or the right–most position, but it cannot be at both. The partitioned fractional tasks c_l and c_r, although allocated to different task sequences, cannot be assigned to two overlapping execution intervals. Or else, the degree of parallelism will increase and violate the maximum degree of parallelism constraint. The match_sequence procedure relaxes the position constraints of fractional tasks, solves the formulated bipartite matching problem, and fixes the conflicts in the maximum matching solution.

The match_integral_task procedure finds the maximum number of integral tasks in adjacent time intervals that can be scheduled consecutively on the same processor. Since each integral task requires a full processor, the maximum number of matches between the integral tasks in adjacent time intervals is predictable. The number of the integral tasks of T^j is $\lfloor x_i^j/t_i \rfloor$ in g_i and $\lfloor x_{i+1}^j/t_{i+1} \rfloor$ in g_{i+1}. The maximum number of matches is $\gamma_i^j = \min(\lfloor x_i^j/t_i \rfloor, \lfloor x_{i+1}^j/t_{i+1} \rfloor)$. The match_integral_task procedure takes γ_i^j integral tasks from the adjacent time intervals and

```
Procedure match_sequence
input: I, TS
output: S    <S is the set of schedules for each processor.>
begin
    match_integral_task(I);
    match_fractional_task(TS);
    fix_conflict(TS, M);    <M is the maximal matching.>
    allocate_processor(Z, M);  <Z is the set of matched integral tasks.>
end;
```

Figure 3.2: Procedure match_sequence

produces γ_i^j matched pairs (or edges) of integral tasks. The remaining integral tasks are added to TS_i.

For the task sequences containing the fractional tasks and leftover integral tasks, we relax the position constraints among the fractional tasks, and formulate and solve them as a bipartite matching problem. The match_fractional_task procedure constructs a bipartite graph $G_i = (TS_i, TS_{i+1}, E_i)$ for each pair of adjacent time intervals as follows. TS_i is the task sequences in the time interval $[a_i, a_{i+1}]$. E_i is a set of edges connecting two task sequences in adjacent time intervals. An edge linking two task sequences is in E_i, if one task sequence is in TS_i, the other task sequence is in TS_{i+1}, and both have a subtask of a task T^j.

Then, the match_fractional_task procedure solves the bipartite matching problem and gives the matched edges in the maximum matching. The task–sequence pairs linked by the matched edges are allocated on the same processors. T^j's subtask in the task sequence in TS_i (respectively, TS_{i+1}) is moved to the right–most (respectively, left–most) position, if it is not already at that position. Hence, a preemption and possibly a migration are saved. As an example, the match_fractional_task produces the bipartite graph depicted in Figure 3.3. A task sequence (T^1 T^2 T^3) is in the time interval [0, 7]. A task sequence (T^3) is in [7, 12]. Since both task sequences have a subtask of T^3, an edge ((T^1 T^2 T^3), (T^3)) is added to E_0. The dotted and solid edges represent E_0, while the solid edges represent the maximum matching.

The fix_conflict procedure detects the conflicts in the solution of the relaxed matching problem, and enforces the position constraints. The violations of the position constraints can be easily detected. First, it is not possible to schedule a task simultaneously to the left–most and right–most positions in a task sequence. Second, the execution intervals of the partitioned fractional tasks c_l and c_r are checked for overlapping. If c_l and c_r overlaps, then c_l and c_r are moved towards the left–most and right–most positions respectively until no overlapping. The total processing time of c_l and c_r is less than t_i. The no overlapping condition will be satisfied

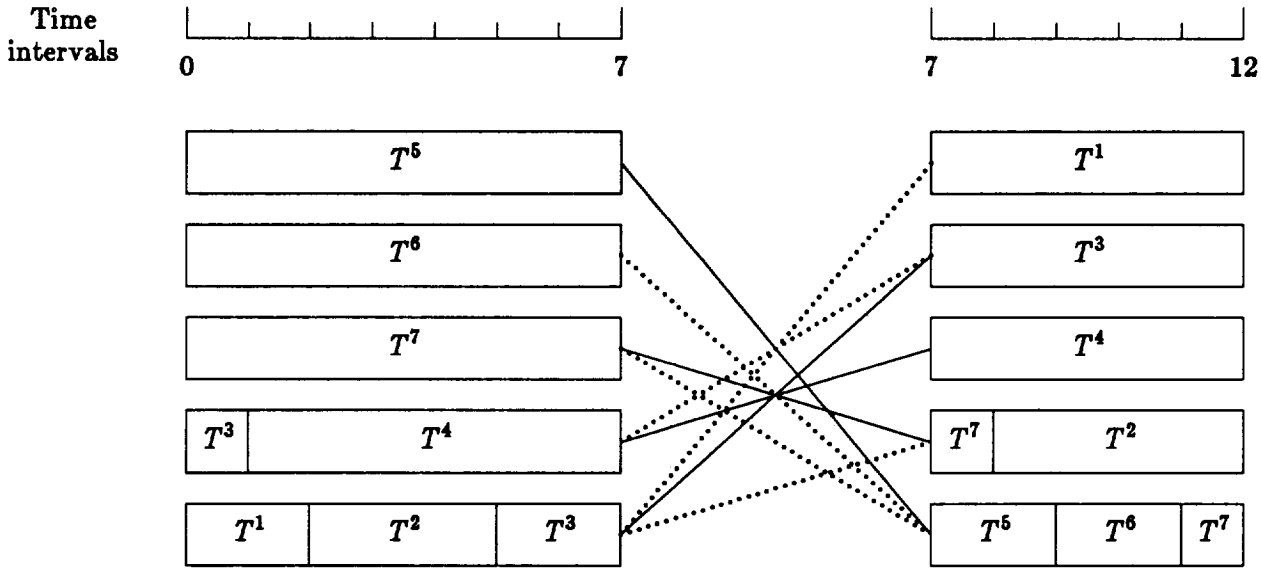


Figure 3.3: A bipartite graph for the example problem

eventually. In either case, one matched edge is deleted. Finally, the allocate_processor procedure assigns the same processors to the task sequences linked by the matched edges.

3.4.3. Bounds on Process Preemptions and Migrations

We have shown by construction that the form_sequence procedure produces a feasible schedule for the tasks $\{x_i^j \mid j=1,2,\ldots,n\}$ and $\sum_{j=1}^{n} x_i^j \leq mt_i$ on m processors in the time interval $[a_i, a_{i+1}]$. In this section, we discuss the upper bounds on the number of process preemptions and on the number of process migrations. These bounds can be construed as linear functions and incorporated in the linear programming formulation.

Preemptions are either the suspension or resuming of a process. Since a context switch consists of a suspension and a resuming, a half of the context switch cost is charged to the suspended process and the other half to the resuming process. A process migration is needed if an execution thread is moved from one processor to the others. For a task T^j, $\max_{i=1}^{l}\lceil x_i^j/t_i \rceil$ threads are spawned. Although migrations imply preemptions, they are counted separately. Preemptions and migrations can occur either between adjacent time intervals (inter–interval) or in a time interval (intra–interval).

Since the integral tasks are not preempted and migrated in a time interval (intra–interval), we need to count the fractional tasks only. The fractional_task_seq generates at most two preemptions and one migration for each partitioned fractional task. There are at most b^j fractional tasks for T^j. b^j is the number of x_i^j of T^j included in the linear programming formulation. Therefore, the number of preemptions generated for T^j is bounded from above by $2b^j$. The number of migrations generated for T^j is bounded from above by b^j. By the same token, the form_sequence procedure generates at most $2(m-1-\sum_{j=1}^{n} y_i^j)$ preemptions in the time interval g_i. The number of migrations in the time interval is bounded from above by $m-1-\sum_{j=1}^{n} y_i^j$. We show that these bounds are tight by the problem instance shown in Table 3.3. Figure 3.4 depicts the optimal schedule for the problem with the minimum preemption and migration overhead. Both the number of preemptions and the number of migrations for $T^2, T^3, \ldots,$ and T^m achieve the upper bounds. So do the number of preemptions and the number of migrations in the time interval [0, m+1].

We count the numbers of preemptions and migrations between adjacent time intervals (inter–interval) next. The number of migrations between adjacent time intervals for each task cannot exceed $m/2$. The match_integral_task procedure guarantees to produce the γ_i^j or $\min(\lfloor x_i^j/t_i \rfloor, \lfloor x_{i+1}^j/t_{i+1} \rfloor)$ matched edges in adjacent time intervals for task T^j. The number of

Table 3.3: A problem to show the tight bound

	r	d	τ	h	s
T^1	0	m + 1	m	m	0
T^2	0	m + 1	m	m	0
		...			
T^{m+1}	0	m + 1	m	m	0

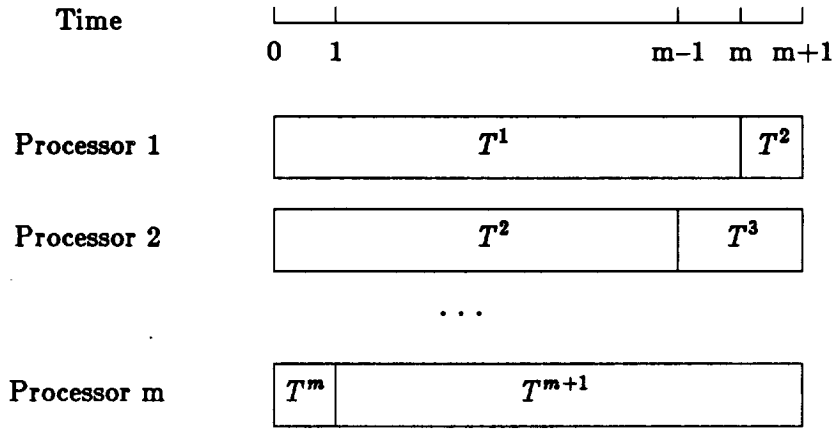


Figure 3.4: The optimal schedule

migrations incurred by T^j between $[a_i, a_{i+1}]$ and $[a_{i+1}, a_{i+2}]$ is bounded from above by $\lceil x_i^j/t_i \rceil - \gamma_i^j$. $|x_i^j/t_i - x_{i+1}^j/t_{i+1}| \leq |\lceil x_i^j/t_i \rceil - \lfloor x_{i+1}^j/t_{i+1} \rfloor| < |x_i^j/t_i - x_{i+1}^j/t_{i+1}| + 2$. Removing the floor and ceiling operator, we get the linear bound $\beta_i^j = x_i^j/t_i - \min(x_i^j/t_i, x_{i+1}^j/t_{i+1}) + 2$. Since each migration requires two preemptions for T^j, the number of preemptions is at most $2\beta_i^j$. These upper bounds are not tight, however.

The complexity of our algorithm is $O(n^2 \log n + 1/\epsilon^3)$ where ϵ is the degree of accuracy or the number of bits required to encode a number. With more computational efforts, our

algorithm can be improved to reduce more preemptions. For instance, we can find a set of task sequences $\{S_k | \sum_{f_i \in S_k} f_i = 2t_i\}$ in F_i using the algorithm for the sum of subset problem. Each task sequence is decomposed to two sequences of equal processing time. The fractional_task_seq procedure selects tasks for a task sequence in the first–fit manner. The first–fit selection may affect future decisions and thus lead to suboptimal results.

The time complexity of our algorithm is pseudo–polynomial in time scale (ϵ). If ϵ is small, our algorithm can be computational expensive. An inexpensive approach is to use McNaughton's rule which has a linear time complexity, $O(n)$ [61]. McNaughton's rule, however, does not try to reduce the number of partitioned fractional tasks. Hence, there is a trade–off. If ϵ is not too small and it is important to reduce the numbers of preemptions and migrations, our algorithm can be used. Otherwise, the simple McNaughton's rule can be used.

The following example illustrates the algorithm described in this section. Given 10 tasks whose parameters are shown in Table 3.4, we want to construct a five processor schedule with the minimum total weighted error. Suppose the multiprocessing overhead function is $o^j \sum_{i=1}^{l-1} y_i^j$. As discussed in Section 3.3, we formulate and solve it as a linear programming problem. The solution is given in Table 3.5. The minimum total weighted error is 144.18. Figure 3.5 depicts a feasible schedule constructed by the described procedures from the linear programming solution in Table 3.5.

3.5. Examples of Parallelizable Tasks

The multiprocessor schedule constructed in Section 3.4 allows the degree of concurrency of a parallelizable task to change in a time segment or from a time segment to the next. In other words, the number of processors assigned to the task may vary during its execution. Although such a multiprocessor schedule may be acceptable for some parallelizable tasks, it may not be desirable for the others. We discuss this problem next.

Table 3.4: An imprecise multiprocessor scheduling problem

	r	d	τ	h	s	w	o	K
T^1	2	10	26	10	16	2	0.2	5
T^2	0	19	40	22	18	1	0.4	5
T^3	0	4	8	6	2	3	0.2	5
T^4	10	30	36	10	26	1	0.6	5
T^5	2	5	10	7	3	3	0.2	5
T^6	0	28	60	30	30	1	0.4	5
T^7	10	19	22	15	7	3	0.6	5
T^8	0	9	15	6	9	2	0.2	5
T^9	20	30	24	10	14	2	0.2	5
T^{10}	5	14	11	11	0	3	0.4	5

Parallelizable tasks can be classified into three types according to the dependence relations among the concurrent subtasks. A parallelizable task of the first type is composed of independent concurrent subtasks which have no dependence relation. A parallelizable task of the second type has regular dependence patterns among its concurrent subtasks. A parallelizable task of the third type has irregular dependence relations among its concurrent subtasks and requires complex synchronizations.

The parallelizable tasks of the first type have independent concurrent subtasks. These parallelizable tasks are commonly found in the programs for scientific computations [43]. The designers of parallelizing compilers call them DOALL or FORALL loops [65]. For example, the inner products, vector additions, and vector multiplications in the matrix–vector and matrix–matrix operations can be computed simultaneously. Using the folding technique, we can decompose a FFT into a number of smaller independent FFT's [24]. These independent subtasks can be scheduled to the time segments which may have different degrees of parallelism [60]. The parallelizable tasks of the first type have (near–) linear speedups. Their multiprocessing

Table 3.5: The linear programming solution for the problem in Table 3.4

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
a_i	0	2	4	5	9	10	14	19	20	28
T^1		0	1	7.37	2					
T^2	2	0	1	4	1	9.32	5.23			
T^3	7.43	1.11								
T^4						0	0	1	7	2
T^5		8.89	2							
T^6	0	0	0	0.18	1	0.46	0	4	26.49	
T^7						4	19.77			
T^8	0.57	0	1	4.45						
T^9									6.51	8
T^{10}				4	1	6.22				

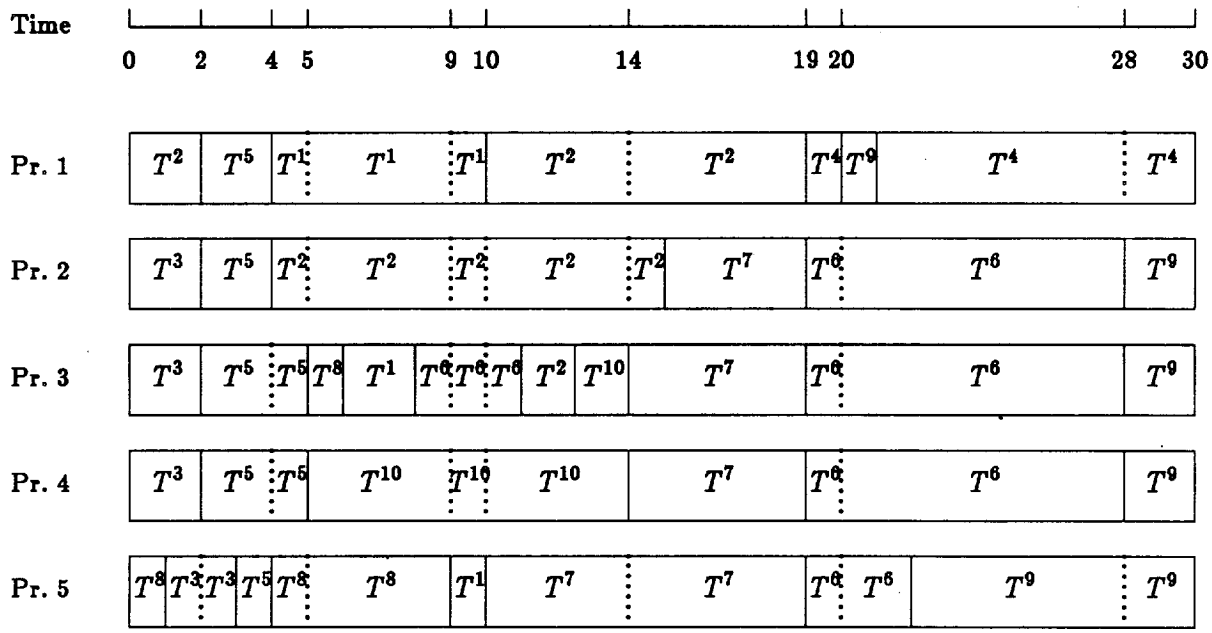


Figure 3.5: A feasible five processor schedule for the problem instance in Table 3.4

overheads can be bounded closely from above by linear functions.

The parallelizable tasks of the second type have regular dependence patterns among the concurrent subtasks. They are known as DOACROSS loops, AND trees, or OR trees [91]. For example, the $(i+k)^{th}$ relation in a recurrence relation is dependent on the i^{th} relation. Because k is usually a constant, the concurrent subtasks (DOACROSS loops) are said to have a constant data dependence distance [65]. Other examples include the parallelizable tasks that have tree–shape dependence patterns such as quick sort, Fibonacci function, divide and conquer algorithms (AND trees), and B&B algorithms (OR trees). Several algorithms can be used to schedule the concurrent subtasks [19, 60]. These parallelizable tasks have good speedups. Their multiprocessing overheads can be bounded from above by linear functions.

The parallelizable tasks of the third type have irregular dependence relations among the concurrent subtasks and require complicated synchronizations. Examples include fault–tolerant algorithms and CAD/CAM algorithms. They usually have high multiprocessing overheads and poor speedups, when they are multiprocessed. Moreover, many efficient parallel algorithms designed for these tasks assume that the number of processors allocated to the algorithms is fixed during the execution [64]. If the number of processors is allowed to vary, the execution speed of the parallel algorithms may be reduced. The multiprocessor overheads may be increased. Therefore, it is best to execute these parallelizable tasks sequentially. Namely, set K^j equal to 1. This may not be acceptable, if the sequential execution time of a parallelizable task T^j is longer than its required response time $(d^j - r^j)$ and parallel processing is the only way for it to produce an acceptable result. There are two approaches to ensure that T^j has a constant degree of concurrency in the multiprocessor schedule and reasonable multiprocessing overheads.

In the first approach, the task T^j is assigned with a rectangular space–time area in each time segment contained in its life time $[r^j,\ d^j]$. The width of the rectangle is the degree of concurrency $(z^j = y^j + 1)$ of T^j where the degree of parallel execution overhead y^j is the same for all time segments in its life time and is restricted to an integer. The length is the assigned

processing time (x_i^j) divided by the degree of concurrency (z^j). A 2–dimensional packing problem is formulated for each time segment. Since the 2–dimensional packing problem is NP–hard [16], we can only hope for an approximate algorithm with a good absolute performance bound for R. R is the ratio of the worst–case make–span to the low bound (instead of the optimal) make–span. The low bound make–span is derived by the sum of rectangular areas divided by m. Note, this is different from the usual definition of R. We then incorporate R in the integer linear programming (*ILP*) formulation of the time allocation problem. ILP is necessary because y^j is restricted to an integer. The inequality constraints of ILP are modified as shown below.

$$\epsilon^j + \sum_{i=1}^{l-1} x_i^j = r^j + \sum_{i=1}^{l-1} o^j y^j \qquad j = 1, 2, \ldots, n$$

$$\sum_{i=1}^{l-1} x_i^j \geq h^j + \sum_{i=1}^{l-1} o^j y^j \qquad j = 1, 2, \ldots, n$$

$$m t_i / R \geq \sum_{j=1}^{n} x_i^j \qquad i = 1, 2, \ldots, l-1$$

$$(K^j - 1) \geq y^j \geq 0 \qquad i = 1, 2, \ldots, l-1. \; j = 1, 2, \ldots, n$$

$$y^j \geq (x_i^j / t_i - 1) \qquad i = 1, 2, \ldots, l-1. \; j = 1, 2, \ldots, n$$

$$x_i^j \geq 0 \qquad i = 1, 2, \ldots, l-1. \; j = 1, 2, \ldots, n$$

$$y^j \text{ is integer} \qquad j = 1, 2, \ldots, n$$

A number of approximate algorithms have been designed for the 2–dimensional packing problem. [16, 80] Blazewicz et al. studied a special case in which the degree of concurrency is either 1 or k where k is a constant greater than 1 [10]. One absolute performance bound for R is 2.5 [16]. It means that only 40% of the processing time in the time seqment g_i is utilized.

In the second approach, we pre–allocate integral subtasks to the parallelizable task T^j. We need to determine a priori which time segments in the life time of T^j are assigned with integral subtasks and the number of integral subtasks assigned to these segments. Since T^j has a constant degree of concurrency, the number (z^j) of integral subtasks assigned to each segment is identical. The total amount of processing time assigned to the task T^j should be no less than the

sum of the processing time (h^j) of its hard task and its multiprocessing overhead. T^j is excluded from the LP formulation. For example, suppose z^j integral subtasks are allocated to T^j in the time segment g_i. We pre–allocate $x_i^j = z^j \times t_i$ processing time to T^j in g_i. The third constraint in the LP formulation is changed to $mt_i - x_i^j \geq \sum_{k=1}^{n} x_i^k$ where j is not in [1, n]. The rest of the LP formulation remains the same. The integral subtasks of T^j are added to the input parameter I of the procedure match_seq in the second phase.

The second approach has four advantages over the first approach. First, there exists efficient polynomial–time algorithms for solving LP's but none for ILP's. Second, it is sufficient but not necessary for a task to have a rectangular space–time area in a time segment. The necessary requirement is that the number of processors assigned to the concurrent tasks of a parallelizable task is constant. Yet, the concurrent subtasks may migrate to different processors. Third, the time segments have less idle processing time and thus are better utilized. Fourth, if the task T^j have integral subtasks assigned to the adjacent time segments, they are not migrated and preempted in these segments. This is assured by the match_integral_task procedure in the second phase. However, the second approach has two shortcomings. First, we must decide a priori the amount of processing time to assign to T^j and in which time segments. Second, T^j has only integral subtasks in its life time.

3.6. Summary

We have presented an approach for scheduling parallelizable real–time tasks on multiprocessors. These real–time tasks are flexible and can be parallelized to several versions each of which has a different degree of parallelism and a multiprocessing overhead. Our approach is based on linear programming. It guarantees to find, if one exists, the optimal time allocation with the minimum total weighted error. Efficient algorithms for linear programming problems are used to solve the time allocation portion of the imprecise multiprocessor scheduling problem. We have also shown a pseudo–polynomial time algorithm for constructing a preemptive

multiprocessor schedule from the linear programming solution. The algorithm reduces the number of process preemptions and migrations which are discrete and nonlinear. More importantly, the algorithm guarantees the number of process preemptions and the number of migrations generated in the multiprocess schedule not to exceed some linear upper bounds.

The linear programming formulation of the imprecise multiprocessor scheduling problem is simpler and more intuitive than the previously proposed (minimum cost maximum flow) network flow formulation [77]. The linear programming formulation solves the imprecise multiprocessor scheduling problem in one step, whereas the network flow formulation requires two steps and a change of parameter values after the first step. Moreover, it is difficult to introduce multiprocessing overheads in the network flow formulation, even if the overheads are constants. The complexity of the network flow approach is $O(n^6)$, comparable to the complexity of the linear programming approach.

Chapter 4
Scheduling of Replicated Periodic Tasks

This chapter presents two algorithms for scheduling replicated periodic tasks on a multiprocessor system under fault–tolerant requirement. In Section 4.1, we describe our approach. Section 4.2 describes the related research in scheduling real–time tasks on a multiprocessor system. We describe the system and failure model assumed and define formally the *replicated imprecise task allocation* problem in Section 4.3. Sections 4.4 and 4.5 each describe one of the two algorithms. Section 4.6 presents the results of the stochastic analysis and experimental evaluation of both algorithms. We give a summary in Section 4.7.

4.1. Our Approach

Two basic approaches to tolerating computer component failures are replication and masking, and roll–back and retry [7, 17]. For time–critical computer applications such as hard real–time systems, replication and masking is more appropriate than roll–back and retry which usually requires more time for error recovery.

A general approach to tolerating the loss of computational resources is to leave less critical tasks unprocessed. For example, fixed priority preemptive scheduling algorithms such as the rate monotonic algorithm (*RM*) may not process lower priority tasks [57]. In RM, a task with a shorter period is assigned with a higher priority. A period transformation method was proposed to change task periods so that the priority or criticality of a task can be assigned independently of its period [74].

In the thesis, we manage the loss of computational resources by the imprecise computation model. The model trades off the quality of the result produced by a task with the amount of processing time needed to produce it. Despite the loss of some computational resources, the imprecise computation model can be used to produce a lower quality yet acceptable result with less

processing time. Hence, the imprecise model provides hard real–time systems with flexible functionality and permits their performance to degrade gracefully in case of hardware failures.

The task system of the replicated imprecise task allocation problem consists of n periodic tasks each of which can be logically decomposed into a hard task and a soft task. Aperiodic and sporadic tasks can be processed by periodic task servers [82]. Each periodic task maybe replicated. The copies or *clones* of a task must be assigned to distinct processing elements (*PE*'s). The clones allocated to a PE are scheduled by the priority–driven scheduling algorithms (*PDSA*), i.e., RM or the earliest deadline first algorithm (*EDF*) [57]. The processor utilization of a PE is the sum of the utilization (processing time / period) of the clones allocated to the PE. The schedulability threshold is the maximum processor utilization of a PE below which the scheduling algorithm can guarantee all deadlines of the periodic clones allocated to the PE. The threshold is determined by the algorithm used. The objective of our algorithms is to find for each periodic task (a) the amount of its processing time and (b) the assignment of its clones to the PE's so that the processor utilization of every PE is not more than its schedulability threshold and the total weighted utilization (TWU) of the task system is maximized. We have shown in Section 2.3 that the optimization problem can also be formulated as minimizing the total weighted unscheduled portions of the soft tasks in a task allocation [58].

Liu and Layland have shown that the RM algorithm is optimal among all fixed priority scheduling algorithms for periodic task systems [57]. The EDF algorithm is optimal among all dynamic priority scheduling algorithms. These algorithms are optimal in the sense that they can always produce a feasible schedule if any other scheduling algorithm of the same scheme (i.e., fixed priority or dynamic priority) is able to do so. RM assigns a lower priority to a task with longer period. Since periods are fixed, periodic tasks have fixed priority. EDF always executes the task with the earliest deadline. Since periodic tasks have deadlines occurring periodically, their assigned priorities may vary from request to request. When the algorithm is RM, the threshold of a PE is $c(2^{1/c}-1)$ where c is the number of clones allocated to the PE. When the

algorithm is EDF, the threshold of every PE is one.

We design the *allocate* algorithm and the *modify* algorithm for solving the replicated imprecise task allocation problem. The allocate algorithm seeks a good initial assignment of tasks to PE's, while the modify algorithm finds a new task assignment with the least changes from the old one as PE's fail or recover. Both algorithms have three phases. In the first phase, a greedy algorithm computes the upper bound of TWU and the upper bound utilization of each task. The greedy algorithm derives these upper bounds by removing some constraints in the original problem. In the second phase, each algorithm calls different subroutines with the upper bound task utilizations to generate task allocations on the PE's. Because the upper bounds are used, it is likely that the processor utilizations of some PE's exceed the schedulability thresholds. It is necessary in the third phase to adjust the task utilizations so that every processor utilization is no more than the schedulability threshold and TWU is maximized. We formulate the problems as linear programs (LP) and solve them using known efficient LP algorithms.

The difference between the allocate algorithm and the modify algorithm lies in the second phase in which different allocation routines are called to produce desirable task allocations. The desirable task allocations for the allocate algorithm are those close to the optimal ones. We have used two different subroutines to generate two different task allocations. We then choose the one which is closer to the optimal task allocation. For the modify algorithm, the desirable task allocations are those maintaining as much location continuity as possible with respect to the previous task allocation before PE failures. To achieve this, the allocation subroutine called by the modify algorithm will not re–allocate those clones already assigned to the fault–free PE's so that these clones are not migrated. Under this constraint, the modify algorithm tries to maximize TWU.

4.2. Related Research

There are two basic approaches to scheduling periodic tasks on a multiprocessor system. The *partitioning* method divides the task set into groups and assigns each group to a distinct PE. The group of tasks assigned to a PE can then be scheduled by the techniques for a single PE [57]. The other approach is the *non-partitioning* method. It treats the entire collection of PE's as a large virtual PE. The periodic tasks are scheduled dynamically on all PE's, when the requests for executing them arrive.

The non-partitioning method may lead to very low processor utilizations for some task sets when they are scheduled on multiple PE's using RM [18, 67]. Kim and Welch use a non-partitioning load balancing scheme for the applications where rollback-and-retry is an acceptable mode of recovery. They show that the recovery blocks can be executed in a distributed manner with low overheads [39]. The distributed execution of recovery blocks is capable of masking both hardware and software failures. When a hardware failure occurs during the execution of a task, the recovery block or alternate version of the task is retried later and executed on a different PE. When a software failure is uncovered in a task, the recovery block of the task is executed on any non-faulty PE. If successful, the result produced by the recovery block replaces the erroneous result produced by the primary version as the output of the task.

An extremely stringent reliability would require a large number of (primary) clones which may swamp the system and cause it to have long response times. Krishna and Shin suggest to change some primary clones to ghost clones which are backup copies lying dormant until they are activated to take the place of corresponding primary clones or previously activated ghost clones whose processors have failed [42]. They present an algorithm that ensures that backup schedules can be efficiently embedded within the primary schedule to ensure that hard deadlines continue to be met. Both primary and ghost clones are assigned processing time in the primary schedule. Although the ghost clones are initially passive and do not in that state require any processing time, they represent a latent demand for processor time that must be allowed for. For this

reason, they affect the scheduling of the primary clones, and tend to lower the efficiency of the primary schedules. Moreover, the algorithm calls an optimal task allocation algorithm and an optimal task scheduling algorithm which they assume are given.

Ramamritham considers a very general multiprocessor task scheduling problem with fault tolerance, communication network, memory, and other resource constraints [68]. A partitioning heuristic rule is designed for allocating periodic tasks to PE's. The heuristic rule groups two communicating tasks together and assigns them to the same PE, if the amount of communication between them is above a tunable threshold and they are not replicas of the same task. Once the allocation of tasks is determined, the multiprocessor scheduling problem is simplified to that of a single PE. Again, several heuristic rules are designed to speed up the search for a feasible schedule.

Chung et al. use the partitioning algorithms designed by Dhall and Liu for allocating imprecise periodic tasks to PE's [14]. They propose several optimal and heuristic algorithms for scheduling imprecise periodic tasks on a single PE [77]. However, Dhall and Liu's algorithms do not ensure that the clones of a task are assigned to distinct PE's.

The partitioning approach taken by Bannister and Trivedi is the closest to our approach [9]. They design an approximation algorithm for allocating replicated periodic tasks to distinct PE's so that the processor loads of all PE's are balanced. Their analysis shows that the variations in the processor utilizations of any task allocation produced by the algorithm is bounded from above by a small number. In other words, the algorithm is a good load balancing algorithm. We extend their task system to a more general task system. Our algorithm also uses their task allocation algorithm as a subroutine. Hence, their results are derivable from our model, although our objective (maximize TWU) is different from theirs (load balancing).

In Chapter 3, we generalize the non–partitioning method and design an algorithm for scheduling parallelizable imprecise tasks with multiprocessing overheads. A task is parallelizable, if it can be decomposed to a number of concurrent subtasks (clones). The time allocation part of

the problem is formulated as an LP and solved using an efficient LP algorithm. To construct a multiprocessor schedule, we also need to decide which task to run on which processor. The task allocation part of the problem is formulated as the subset sum problem and the bipartite matching problem which are then solved using known efficient algorithms. However, the allocation algorithm may assign the clones (concurrent subtasks) of a task to the same PE. We are currently improving the algorithm so that the clones of a task are assigned to distinct PE's. This work and the work presented in this chapter show a contrast between the partitioning approach and the non–partitioning approach. The partitioning approach may have lower processor utilizations because of the sub–optimal partitioning of periodic tasks, while the non–partitioning approach has the multiprocessing overheads associated with the migration of concurrent subtasks.

4.3. Replicated Imprecise Task Allocation Problem

Before defining the replicated imprecise task allocation problem, we describe two schemes for tolerating different types of hardware failures by replication. The real–time application software is assumed to be error–free. We describe the computer system model and the fault–tolerant schemes next.

For our study, we assume the real–time computer is compose of m identical PE's connected by an interconnection network (Figure 4.1). Each PE has a processor and a local memory. The interconnection network supports multiple physical communication channels between any two PE's. Although PE's may fail maliciously, we assume that the interconnection network supports reliable broadcasts [8, 13, 40]. Let (*operation*) *mode* i characterize the state in which the real–time computer has i non–faulty PE's. So our system has m operation modes. We assume that the operating system has the capability to detect, isolate, and remove these failed PE's from service [17, 21]. In addition, the repair time is much smaller than the mean time between successive component failures [42]. When the allocate algorithm and the modify algorithm are applied at

run time, they will consume computational resources. The execution of these algorithms is modeled as an aperiodic task and scheduled using the methods proposed in [82].

Using our model, two replication resolution schemes can be used. Suppose there are two periodic tasks T^1 and T^2. and the subscript the operation mode. The result produced by T^1 is consumed by T^2. When T^1 and T^2 are replicated, the result produced by each clone of T^1 is broadcast to all clones of T^2. The additional processing time needed for broadcasting is added to the processing time of T^1. In the first scheme, a clone of T^2 may use any one of the results that arrive on time. The choice is left to each clone of T^2. Hence, neither agreement nor synchronization among the clones of T^2 is needed. If T^1 has c clones, then T^1 can tolerate up to $c-1$ crash/omission/timing failures [17].

The second scheme is similar to the first scheme except each clone of T^2 determines its input by a majority vote of the results that arrive on time. Again, neither agreement nor synchronization among the clones of T^2 is necessary. The additional processing time needed for voting is added to the processing time of T^2. If T^1 has c clones and faulty components may produce identical errors, then T^1 can tolerate up to $\lfloor (c-1)/2 \rfloor$ response/arbitrary failures. On the other hand, if faulty components do not produce identical errors, then T^1 can sustain up to $c-2$ response/arbitrary failures.

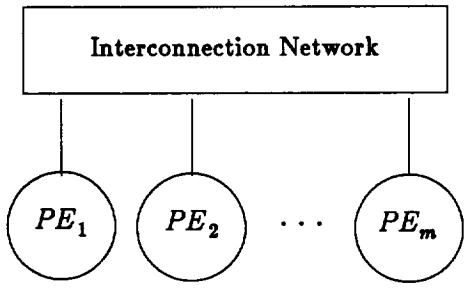


Figure 4.1: Logical organization of the multiprocessor

The reliability requirement and the nature of a real–time fault–tolerant application determine which resolution scheme to use for each task and specifies the number of clones needed for each operation mode. The second scheme does not handle the case in which a clone of T^2 needs the majority result at the deadline of T^1, since some extra time is needed for voting. This problem is not addressed in this chapter. One possible approach is to introduce an earlier deadline at which the clones of T^1 after the data processing can synchronize with one another for voting. The schedulability bounds for periodic tasks with arbitrary deadlines has been studies in the paper [48].

4.3.1. Problem Definition

A replicated periodic task system *RPTS* is composed of a collection of n periodic tasks. Each task T^j may be replicated and has c_i^j identical copies for operation mode i. The number of clones of T^j for mode i is no more than the number of non–faulty PE's (i.e., $c_i^j \leq i$). c_i^j is usually equal to or greater than c_{i-1}^j; thus $<c_m^j, c_{m-1}^j, \ldots, c_1^j>$ is a non–increasing sequence. We use T_i^j to represent the clones $\{C_{i1}^j, C_{i2}^j, \ldots, C_{ic_i^j}^j\}$ of task T^j, and $RPTS_i$ the tasks $\{T_i^1, T_i^2, \ldots, T_i^n\}$ for mode i. Hence, $RPTS_i$ is a multiset.

The replicated imprecise task allocation problem is consisted of m task allocation problems, one for each operation mode. They are referred as the sub–problems of the replicated imprecise task allocation problem. A task allocation for the sub–problem corresponding to mode i is an assignment of the clones in $RPTS_i$ to i identical non–faulty PE's. A task allocation is *feasible*, if

(1) every clone can produce an acceptable result before its deadline,

(2) no two clones of a task are assigned to the same PE,

(3) every clone is executed by exactly one PE at each mode, namely no task migration from one PE to another is allowed, and

(4) the processor utilization of every PE is not more than the schedulability threshold.

A clone is said to produce an acceptable result when its hard task is completed before its deadline. Both the allocate algorithm and the modify algorithm assign an equal amount of the processing time (x_i^j) to each clone of task T_i^j. x_i^j should be no less than the processing time (h^j) of its hard task, but is no more than its total processing time (r^j). The utilization u_i^j of each clone of T_i^j (or the utilization of T_i^j) is x_i^j/p^j.

A feasible task allocation for mode i is optimal, if its total weighted utilization $TWU_i = \sum_{j=1}^{n} w^j u_i^j$ is maximum among all feasible task allocations. Since $i \geq 1$, $w^j > 0$, and $u_i^j \geq 0$, TWU_i of feasible task allocations is greater than 0. We can assume that TWU_i of infeasible task allocations is 0.

When either s^j or c_i^j of each task T_i^j in $RPTS_i$ is zero, we can show that this sub–problem is NP–complete by transforming the NP–complete bin–packing problem to it [25]. This sub–problem is clearly in NP, since a non–deterministic algorithm can guess a task allocation of the hard clones and verify easily the feasibility of the task allocation. The bin–packing decision problem has a finite set I of items, a size $s(j) \in Z^+$ for each $j \in I$, a positive bin capacity B, and a positive integer i. The problem is to find a partition of I into disjoint sets $I_1, I_2, \ldots, I_i$ such that the sum of sizes of the items in each set is B or less. To transform it to a sub–problem of the replicated imprecise task allocation problem, we let $c_i^j = 1$, $h^j/p^j = r^j/p^j = s(j)$, the schedulability threshold be B, and the number of non–faulty PE's be i. The bin–packing problem is thus reduced to an instance of the decision sub–problem corresponding to mode i. Hence, the sub–problem is NP–complete. The replicated imprecise task allocation is at least NP–hard.

4.4. Allocate Algorithm

It is unlikely that there exists an optimal polynomial time algorithm for solving the replicated imprecise task allocation problem. We propose the heuristic allocate algorithm shown in

Figure 4.2 for solving the sub–problem corresponding to operation mode i. The algorithm has three phases.

Phase 1: Greedy Algorithm

Although it is difficult to solve the sub–problem optimally, an upper bound for the total weighted utilization TWU_i^{upper} and an upper bound utilization u_i^j of each task T_i^j in $RPTS_i$ can be computed easily when the scheduling algorithm is EDF. Indeed, if we relax the second and third requirements for feasible task allocations, i.e., if the clones of a task can be on the same PE

```
Algorithm allocate
input: RPTS, i, PDSA;
output: TWU_i;
begin
  ({u_i^j}, TWU_i^upper) = greedy algorithm(RPTS, i, PDSA);      <Phase 1>
  if ((PDSA is EDF) and (u_i^j < h^j/p^j for some j)) then
     Call exception handler;     <No feasible task allocation.>
  else
  begin   <Phase 2>
     A_1 = LUF({u_i^j}, PDSA);   <A_i's are task allocations.>
     A_2 = MF({u_i^j}, PDSA);
     if ((PDSA is EDF) and ((max-proc-util(A_1) ≤ 1) or (max-proc-util(A_2) ≤ 1))) then
           TWU_i = TWU_i^upper;      <The maximum TWU_i found.>
     else
     begin
           A_3 = LUF({h^j/p^j}, PDSA);
           A_4 = MF({h^j/p^j}, PDSA);
           <Phase 3>
           TWU_i = max(LP(A_1, PDSA), LP(A_2, PDSA), LP(A_3, PDSA), LP(A_4, PDSA));
           if (TWU_i = 0) then
              Call exception handler;
     end;
  end;
end;
```

Figure 4.2: The allocate algorithm

or migrate, the upper bounds can be computed by a greedy algorithm. When the scheduling algorithm is EDF, the available processor utilization is i (the number of non–faulty PE's).

When the scheduling algorithm is RM on the other hand, it is difficult to calculate the available processor utilization because it depends on the number of clones assigned to each PE. We assume that the number of clones assigned to a PE can be approximated by the average number of clones per PE, c/i where c is $\sum_{j=1}^{n} c_i^j$. The available processor utilization can be approximated by $c(2^{i/c}-1)$. Hence, when the scheduling algorithm is RM, TWU_i^{upper} and the upper bound utilization of a task may not be absolute upper bounds. The greedy algorithm has two steps.

(1) Each clone in $RPTS_i$ is assigned with the processing time of its hard task. If the scheduling algorithm is EDF and the total utilization of these clones exceeds i, then there is no feasible task allocation and we can stop the algorithm. Otherwise, the greedy algorithm sorts the task in $RPTS_i$ in non–increasing order of w^j/c_i^j.

(2) The algorithm tries to assign an additional s^j processing time to each clone of the first task T_i^j in the sorted task sequence. If the addition of s^j causes the total utilization of all clones to exceed the available processor utilization, then the algorithm tries to assign as much processing time as possible to each clone of T_i^j.[1] In this case, we stop the algorithm. Otherwise, the algorithm repeats the second step for the next task in the sequence.

When the scheduling algorithm is EDF, the greedy algorithm is optimal for solving the relaxed sub–problem due to the following facts. The processing time assigned to each clone in the first step is necessary for a feasible task allocation. The remaining problem after the first step is equivalent to the fractional knapsack problem [31], and the algorithm described in the

[1]Specifically, the processing time allocated to each clone of T_i^j is determined by the remaining processor utilization times p^j divided by c_i^j. The remaining processor utilization is the available processor utilization minus the total utilization of all clones.

second step is optimal. The complexity of the greedy algorithm is $O(n \log n)$.

Phase 2: LUF Algorithm and MF Algorithm

In the second phase, the allocate algorithm calls the *LUF* (largest utilization first) algorithm and the *MF* (multifit) algorithm to generate two task allocations for $RPTS_i$ using the upper bound u_i^j generated in the first phase. LUF has two steps.

(1) It sorts the tasks of $RPTS_i$ in non–increasing order of utilization. When multiple tasks have the same utilization, they are arranged in non–increasing order of the number of clones.

(2) LUF assigns the c_i^j clones of the first task T_i^j in the ordered task sequence to the c_i^j PE's that have the least excesses (or most slacks) in processor utilization. The excess of a PE is its current processor utilization minus its schedulability threshold and may be less than zero. Note that when the scheduling algorithm is RM, its threshold is anticipated as $(c+1)(2^{1/(c+1)}-1)$ where c is the number of clones currently assigned to it. LUF repeats the second step for the next task in the sequence.

The MF algorithm uses a first–fit bin–packing heuristic to minimize the maximum processor utilization. MF has three steps.

(1) It sorts the tasks of $RPTS_i$ in non–increasing order of the number of clones. If tie, the tasks are arranged in non–increasing order of utilization. MF sets the maximum processor utilization to the value, $\sum_{j=1}^{n} c_i^j u_i^j / i$.

(2) MF assigns the c_i^j clones of the first task T_i^j in the ordered task sequence to the c_i^j PE's that fit with the lowest indices. It repeats the second step for the next task in the sequence.

(3) If MF fails to find a feasible packing with the maximum processor utilization, it increases the maximum processor utilization to one half of the current maximum processor utilization plus $4/3 + (c_i - 2)/3i$ where c_i is $\max_{j=1}^{n} c_i^j$. It goes to the second step. If a feasible packing

with the maximum processor utilization is found, MF performs a binary search for the minimum maximum processor utilization in the interval between the current maximum processor utilization and the previous maximum processor utilization.

The interval to be searched in step (3) is between the lower bound value ($\sum_{j=1}^{n} c_i^j u_i^j / i$) and $4/3+(c_i-2)/3i$ where c_i is $\max_{j=1}^{n} c_i^j$. This is because we can show (Theorem 4.1 in Section 4.6.1) that the maximum processor utilization of LUF is bounded from above by $4/3+(c_i-2)/3i$, when the scheduling algorithm is EDF. When MF generates a maximum processor utilization larger than $4/3+(c_i-2)/3i$, the maximum processor utilization is larger than that of LUF. When the scheduling algorithm is RM, the upper bound is lower. Notes that when $c_i^j=1$ for all tasks, MF becomes the generalized FF version of the multifit algorithm.

When the scheduling algorithm is EDF and when either of the two maximum processor utilization associated with the task allocations $\mathbf{A}_1$ and $\mathbf{A}_2$ is 1 or less, TWU_i^{upper} is optimal. In the inequality case, $\sum_{j=1}^{n} c_i^j \tau^j / p^j < i$. Each soft task is executed to completion and there is idle processing time. We can stop the allocate algorithm. Since the upper bound task utilizations are used, the excesses of some PE's are usually positive. We define the *overage* of a task allocation as the largest excess of all PE's. We shall explain later in Section 4.6 that minimizing overage is consistent with maximizing TWU.

Because LUF and MF are heuristic in nature, they may generate infeasible task allocations. To increase the likelihood of getting feasible task allocations, the allocate algorithm also calls LUF and MF withe the hard subtasks of the clones only. Since LUF and MF are effective and h^j is usually less than τ^j, LUF and MF usually can find feasible task allocations for $RPTS_i$. The complexity of LUF is $O(n\log n)$. The complexity of MF is $O(n\log n+in)$ where i is the number of iterations performed in the binary search and is usually preset to a constant.

Phase 3: Linear Programming Formulation

LUF and MF may generate inferior or infeasible task allocations, since the upper and lower bound utilizations of a task are used. Some processor utilizations in the task allocations may be greater than the schedulability thresholds (over–utilized) and others less than the thresholds (under–utilized). We need to adjust u_i^j so that the utilization of every PE is close to but not more than the threshold and TWU_i is maximized. Specifically, the utilizations of some tasks allocated to the over–utilized PE's must be reduced, while some task utilizations should be increased in the under–utilized PE's. The problem of maximizing TWU_i can be formulated as an LP shown below in matrix notation.

$$\mathbf{max\ w\ u}$$
$$\mathbf{A\,u} \leq \mathbf{threshold(A,}\ PDSA\mathbf{)}$$
$$\mathbf{h/p} \leq \mathbf{u} \leq \mathbf{r/p}$$

w is $1 \times n$ row vector of w^j and **u** $n \times 1$ column vector of u_i^j. Each task allocation can be described by a matrix **A** of $\{0, 1\}$ with i rows and n columns. An 1 at the j^{th} column and the k^{th} row denotes that a clone of the task T_i^j is assigned to the k^{th} PE. **threshold**(**A**, *PDSA*) is a $i \times 1$ column vector of the schedulability thresholds for i PE's for the PDSA scheduling algorithm. **h/p** and **r/p** are the notations for $n \times 1$ column vectors of h^j/p^j and r^j/p^j respectively. We use known efficient algorithms for LP to compute the TWU_i for each task allocation [36, 89]. The allocate algorithm returns the largest TWU_i of the four task allocations. These algorithms solves our LP problem in $O(n^3)$ steps. The overall complexity of the allocate algorithm is thus $O(n^3)$.

An Example

We illustrate the allocate algorithm by an example. An instance of the replicated imprecise task allocation problem has 10 periodic tasks whose parameters are shown in Table 4.1. The

real–time computer is a multiprocessor consisting of 6 PE's. We apply the allocate algorithm to find a task allocation for the sub–problem instance corresponding to operational mode 6. The scheduling algorithm employed is EDF. The second and third columns of Table 4.2 show the upper bound x_6^j and u_6^j produced by the greedy algorithm for each task T^j. The upper bound total weighted utilization TWU_6^{upper} is 13.493. When LUF is called with u_6^j, it generates the best of the four task allocations. Table 4.3 shows the task allocation matrix (A_1) generated and the processor utilization of each PE. To compute TWU_6, we formulate and solve the problem as an LP. The efficient LP algorithm adjusts the processing time and utilization of each task as shown in Table 4.2. Every PE except PE 2 is fully utilized after the adjustment (Table 4.3). TWU_6 is 12.3533.

Suppose two PE's become faulty. We apply the allocate algorithm again to find a task allocation of $RPTS_4$ on 4 non–faulty PE's. The greedy algorithm gives the upper bound x_4^j and u_4^j for each task T^j depicted in Table 4.4. TWU_4^{upper} is 11.9783. Table 4.5 shows the task

Table 4.1: An example of periodic task system

j	p	τ	h	w	c_6	c_5	c_4	c_3	c_2	c_1
1	20	4	2	5	3	2	2	1	1	0
2	30	5	3	9	3	3	3	2	2	1
3	50	15	2	4	3	2	1	1	1	0
4	50	20	11	8	5	4	3	2	2	1
5	100	29	10	10	4	4	3	3	2	1
6	100	41	20	2	2	1	1	0	0	0
7	200	47	36	7	2	2	2	2	1	1
8	200	90	25	3	1	1	1	1	0	0
9	300	78	49	1	1	1	1	0	0	0
10	300	79	33	6	4	3	2	2	1	1

Table 4.2: Upper bound utilizations given by the greedy algorithm and the utilizations given by LP and their corresponding assigned processing times

j	Greedy Algorithm		LP	
	x_6	u_6	x_6	u_6
1	4	0.200000	2.56666	0.128333
2	5	0.166667	5	0.166667
3	2	0.040000	2	0.040000
4	16.966667	0.339333	18	0.360000
5	29	0.290000	29	0.290000
6	20	0.200000	20	0.200000
7	47	0.235000	47	0.235000
8	90	0.450000	89.6666	0.448333
9	49	0.163333	66.4999	0.221667
10	33	0.110000	33	0.110000

Table 4.3: The task allocation matrix generated by LUF and the processor utilizations

	j										*Processor Utilization*	
PE	1	2	3	4	5	6	7	8	9	10	Greedy Alg.	LP
1	0	1	1	0	0	0	1	1	0	1	1.001667	1
2	1	1	0	1	1	0	0	0	0	0	0.996000	0.945
3	1	0	0	1	1	0	0	0	1	0	0.992667	1
4	0	0	1	1	1	1	0	0	0	1	0.979333	1
5	0	0	1	1	1	1	0	0	0	1	0.979333	1
6	1	1	0	1	0	0	1	0	0	1	1.051000	1

allocation matrix (A_2) and processor utilizations generated by the MF algorithm for this sub-problem. The adjusted processing time and utilization of each task are displayed in Table 4.4. The utilization of every PE is one (Table 4.5) and TWU_4 is 11.9783 after adjustment. Since TWU_4 is equal to TWU_4^{upper}, TWU_4 is optimal.

Table 4.4: Upper bound utilizations given by the greedy algorithm and the utilizations given by LP and their corresponding assigned processing times

j	Greedy Algorithm		LP	
	x_4	u_4	x_4	u_4
1	2	0.100000	2	0.100000
2	5	0.166667	4.65	0.155000
3	15	0.300000	15	0.300000
4	11	0.220000	11	0.220000
5	29	0.290000	29	0.290000
6	20	0.200000	20	0.200000
7	47	0.235000	47	0.235000
8	83.333321	0.416667	80.33333	0.401667
9	49	0.163333	49	0.163333
10	33	0.110000	40.5	0.135000

Table 4.5: The task allocation matrix generated by MF and the processor utilizations

PE	j 1	2	3	4	5	6	7	8	9	10	*Processor Utilization* Greedy Alg.	LP
1	1	1	0	1	1	0	1	0	0	0	1.011667	1
2	1	1	0	1	1	0	1	0	0	0	1.011667	1
3	0	1	0	1	1	1	0	0	0	1	0.986667	1
4	0	0	1	0	0	0	0	1	1	1	0.990000	1

4.5. Modify Algorithm

When PE failures occur, the clones assigned to the faulty PE's that perform critical functions need to be re–allocated to the non–faulty PE's so that the system remains functional. Since it is difficult to predict at system design time which PE's will malfunction at run time, on–line re–configurations of clones are necessary. One solution, as suggested by the example above, is to use the allocate algorithm to produce a new task allocation. However, the allocate algorithm solves each sub–problem independently. When the system switches to the new task allocation,

there may be too many migrations of clones and data. This is not desirable. The migrations of clones and data may (1) delay or omit the execution of the clones, (2) induce transient overloads in processors and overflows in memories, and (3) incur migration costs.

To reduce the migrations of clones and data, critical clones surviving the PE failures should stay at the assigned PE's. We propose the modify algorithm (Figure 4.3) which maintains the continuity of clone locations with respect to the previous task allocation by fixing the clones already assinged to the non-faulty PE's. Let's first define a few terminologies before describing the algorithm.

```
Algorithm modify
input: RPTS, i, PDSA, A_1, miss, excess;        <A_i's are task allocations.>
output: TWU_i;
begin
   ({u_i^j}, TWU_i^upper) = greedy algorithm(RPTS, i);     <Phase 1>
   if ((PDSA is EDF) and (u^j < h^j/p^j for some j)) then
      Call exception handler;
   else
   begin    <Phase 2>
      A_2 = FCLUF(A_1, {u_i^j}, miss, excess, PDSA);
      if ((PDSA is EDF) and (max-proc-util(A_2) ≤ 1)) then
            TWU_i = TWU_i^upper;        <The maximum TWU_i found.>
      else
      begin
            A_3 = FCLUF(A_1, {h^j}, miss, excess, PDSA);
            TWU_i = max(LP(A_2, PDSA), LP(A_3, PDSA));     <Phase 3>
            if (TWU_i = 0) then
               Call exception handler;
      end;
   end;
end;
```

Figure 4.3: The modify algorithm

We call those clones allocated to the non–faulty PE's the *existing* clones; the existing task allocation ($\mathbf{A}_1$) denotes the allocation of these clones. The tasks in $RPTS_i$ are partitioned into three collections. Those have *missing* clones to be added to the existing task allocation, when the number of existing clones of a task T^j is less than c_i^j. Those have *excessive* clones to be removed from the existing task allocation, when the number of existing clones of a task T^j is more than c_i^j. The third collection consists of the remaining tasks. If a task T^j has missing clones, the constraint that the clones are assigned to distinct PE's may limit the choices of assignments. The number of choices is $\binom{i-e_i^j}{c_i^j-e_i^j} = \frac{(i-e_i^j)!}{(c_i^j-e_i^j)!\,(i-c_i^j)!}$ where e_i^j is the number of existing clones of T^j and $c_i^j - e_i^j$ is the number of missing clones of T^j. If a task T^j has excessive clones, the choices of removing these clones may be constrained by the non–faulty PE's that have a clone of T^j. The number of choices is $\binom{e_i^j}{e_i^j-c_i^j} = \frac{e_i^j!}{(e_i^j-c_i^j)!\;c_i^j!}$ where $e_i^j - c_i^j$ is the number of excessive clones of T^j.

The modify algorithm calls the greedy algorithm in the first phase to compute a new TWU_i^{upper} and a new upper bound utilization u_i^j for each task in $RPTS_i$. The new u_i^j may change the processor utilizations of some PE's. In the second phase, the modify algorithm calls the *FCLUF* (Fewest Choice Largest Utilization First) algorithm to add the missing clones to and to delete the excessive clones from the existing task allocation. The clones already assigned to the non–faulty PE's need no re–allocation and will stay at the same PE's in the new task allocation. As explicated in Section 4.4, if the scheduling algorithm is EDF and the maximum processor utilization of all PE's is 1 or less, then TWU_i^{upper} is optimal. To increase the probability of finding a feasible task allocation, we use h^j/p^j instead of u_i^j and call FCLUF again to get another task allocation. In the third phase, an efficient LP algorithm computes the TWU_i's of the two task allocations. The larger TWU_i is returned. FCLUF has two steps.

FCLUF Algorithm

(1) It sorts the tasks in $RPTS_i$ in non–decreasing order of the number of choices. If tie, the tasks are arranged by non–increasing order of utilization.

(2) The second step is either the addition or removal of clones from the existing task allocation. When the first task T_i^j in the ordered task sequence has $c_i^j - e_i^j$ missing clones, FCLUF finds $c_i^j - e_i^j$ PE's that have no clone of the same task and have the least excesses in processor utilization. It adds a clone of T_i^j to each of these PE's. The calculation of excesses is the same as that in LUF. When the first task T_i^j in the ordered sequence has $e_i^j - c_i^j$ excessive clones, FCLUF finds $e_i^j - c_i^j$ PE's that have a clone of T_i^j and have the most excesses in processor utilization. From each PE, it removes a clone of T_i^j. The second step is repeated with the next task in the ordered sequence.

FCLUF allocates the task with the fewest choices of allocation first for the following reasons. If a task has the fewest choices, then it is more likely that all choices are bad. Specifically, the constraint that no two clones of a task are assigned to the same PE may force the missing clones to be added to the PE's with the most excesses in processor utilization. Analogously, the excessive clones may be removed from the PE's with the least excesses in processor utilization. When a task has only one choice, it is best that we add the missing clone to or delete the excessive clone from the destined PE first. In this way, FCLUF reduces the impacts of the bad task assignments on the quality of the solution.

When the modify algorithm is used, the removal and addition of clones can be carried out gradually and incrementally. Recall that the execution of the allocation algorithm and the modify algorithm is modeled as an aperiodic task. Whenever the aperiodic task is assigned computational resources, the excessive clones are removed first from the existing task allocation to avoid the overloading of the PE's. Then, the missing clones are added to the existing task allocation. Note that we have assumed that the mean time between successive failures is much longer

than the repair time. The repairs, though incremental, will be completed before the next failure. Moreover, the modify algorithm can be used to increase the number of clones and hence the reliability of the real–time system, when failed PE's are repaired and ready for service.

An Example

We illustrate the modify algorithm by the same example problem. The allocate algorithm is used to compute the initial task allocation for mode 6 shown in Table 4.3. The modify algorithm changes the initial and subsequent task allocations according to the failed PE's. Suppose PE 4 and 5 become faulty. The modify algorithm calls the greedy algorithm to compute the number of choices of allocations and the upper bound x_4^j and u_4^j for each task T^j displayed in Table 4.6. TWU_4^{upper} is 11.9783. The blank entries in the number of choices column denote that neither addition nor deletion of clones are necessary. The FCLUF algorithm fixes the existing clones and produces two task allocations. Table 4.7 shows the better task allocation ($\mathbf{A}_2$) and the processor utilizations. An efficient LP algorithm adjusts the processing time and utilization of each task so that the TWU_4 of the task allocation is maximized. The last two columns of Table 4.6 display the adjusted processing time and utilization of each task. The utilization of every PE is 1 except PE 2 (Table 4.7) and TWU_4 is 11.34 after adjustment.

4.6. Performance Evaluation

The allocate algorithm and the modify algorithm are evaluated using stochastic analysis techniques and Monte Carlo simulations. We focus the analytical evaluation on the LUF algorithm where the scheduling algorithm is EDF. A sub–problem of a replicated imprecise task allocation problem can have many task allocations. Associated with each task allocation is a TWU_i. Suppose there are two task allocations for a sub–problem. In general but not always, the task allocation with a smaller overage tends to have a higher TWU_i than the task allocation with a larger overage. A larger overage means that the processor utilization of some PE in a task allocation exceeds its schedulability threshold by a greater amount. When the scheduling algorithm

Table 4.6: Upper bound utilizations given by the greedy algorithm and the utilizations given by LP and their corresponding assigned processing times

j	*Number of choices*	Greedy Algorithm		LP	
		x_4	u_4	x_4	u_4
1	3	2	0.100000	2.53333	0.126667
2		5	0.166667	5	0.166667
3		15	0.300000	15	0.300000
4		11	0.220000	11	0.220000
5	2	29	0.290000	29	0.290000
6	4	20	0.200000	20	0.200000
7		47	0.235000	42.66667	0.213333
8		83.333321	0.416667	42	0.210000
9		49	0.163333	49	0.163333
10		33	0.110000	33	0.110000

Table 4.7: The task allocation matrix generated by FCLUF and the processor utilizations

	j										*Processor Utilization*	
PE	1	2	3	4	5	6	7	8	9	10	Greedy Alg.	LP
1	0	1	1	0	0	0	1	1	0	1	1.228333	1
2	1	1	0	1	1	0	0	0	0	0	0.776667	0.803333
3	1	0	0	1	1	1	0	0	1	0	0.973333	1
4	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0
6	0	1	0	1	1	0	1	0	0	1	1.021667	1

is EDF, the overage of a task allocation is its maximum processor utilization minus one. It is more difficult for an LP algorithm to reduce the larger overage to zero so that the task allocation is feasible and without compromising TWU_i. Hence, the task allocation with a smaller overage is preferred. We show that the ratio $R(LUF)$ between the maximum processor utilization produced by LUF and the optimal (minimum) maximum processor utilization for mode i is bounded from above by $4/3+(c_i-2)/3i$ in the worst case, where $c_i=\max_{j=1}^{n} c_i^j$. As the number of tasks increases,

the maximum processor utilization generated by LUF approaches the optimal maximum processor utilization asymptotically. The latter result is derived under a mild assumption on the probability distribution (F) of the utilizations of the periodic tasks. The proofs of the following theorems can be found in the appendix A.

4.6.1. Bounds on the Maximum Processor Utilization of LUF

LUF algorithm sorts the tasks in $RPTS_i$ so that for all $k > j$, either $u_i^k > u_i^j$, or $u_i^k = u_i^j$ and $c_i^k \geq c_i^j$. Let $Z_k^j(LUF)$ denote the processor utilization of the k^{th} PE after the LUF algorithm have scheduled the clones of task T^j. We use $Z_{(1)}^j(LUF)$ and $Z_{(i)}^j(LUF)$ to denote the PE's with the smallest and largest processor utilizations respectively in the ordered sequence of $\{Z_k^j(LUF) \mid k = 1, 2, \ldots, i\}$. $R(LUF)$ is defined as $Z_{(i)}^n(LUF)/Z_{(i)}^n(OPT)$ where $Z_{(i)}^n(OPT)$ is the optimal maximum processor utilization.

Theorem 4.1: $R(LUF) \leq 4/3 + (c_i - 2)/3i$ where $c_i = \max_{j=1}^{n} c_i^j$.

In the example shown in Table 4.1, $c_6 = 5$ and $R(LUF) \leq 3/2$ for mode 6. $R(LUF)$ is always less than or equal to 5/3 because $c_i \leq i$. Let $D^j(LUF)$ denote the difference between the largest processor utilization and the average processor utilization after the clones of the task T^j have been allocated by the LUF algorithm. $D^j(LUF) = Z_{(i)}^j(LUF) - \sum_{k=1}^{i} Z_k^j(LUF)/i$.

Theorem 4.2: If the expected utilization Eu is finite and the probability distribution F of u_i^j is strictly increasing on $(0, \epsilon)$ for some $\epsilon > 0$, then

$$\lim_{n \to \infty} D^n(LUF) = 0. \text{ (a.s.)}$$

Corollary 4.3: If the expected utilization Eu is finite and the probability distribution F of u_i^j is strictly increasing on $(0, \epsilon)$ for some $\epsilon > 0$, then

$$\lim_{n\to\infty}(Z^n_{(i)}(LUF)-Z^n_{(i)}(OPT))=0.\ \text{(a.s.)}$$

Proof: The proof is similar to that in the reference [23]. □

4.6.2. Simulation Results

These analytic results do not show how large an n will make TWU_i close to the optimal, and how TWU_i is affected by RM, the variations in the hard tasks, and the number of PE failures. Furthermore, the analytic results do not show how these variations cause the allocate algorithm and the modify algorithm to produce infeasible task allocations. We address these questions by simulations next.

In each simulation, the performance of the modify algorithm is compared with that of the allocation algorithm with respect to the same the scheduling algorithm. Recall that the sub–problem corresponding to operational mode m is always solved by the allocate algorithm. The sub–problems corresponding to those operational modes less than m can be solved either by the allocate algorithm or the modify algorithm. Because the modify algorithm maintains the continuity of locations, its performance is expected to be inferior than the allocate algorithm. In other words, the modify algorithm should generate more infeasible task allocations than the allocate algorithm. The task allocations produced by the modify algorithm are expected to have lower TWU's than those produced by the allocate algorithm.

The task systems are generated randomly for each simulation. For the lack of prior knowledge, random variables are assumed to be uniformly distributed. The number c^j_m of clones for each task T^j_m in $RPTS_m$ is uniformly distributed in the interval between 1 and m inclusively $[1, m]$. So, T^j_m has an average of $\bar{c}_m(=(1+m)/2)$ clones. The average workload of a hard task $\bar{u}_h=\bar{h}/\bar{p}$ is half of the average task workload $\bar{u}=\bar{\tau}/\bar{p}$. Hence, the average hard workload of $RPTS_m$ is $n\bar{c}_m\bar{u}_h$. We define the average *normalized hard workload* (of $RPTS_m$) as $n\bar{c}_m\bar{u}_h/m$.

We use the following three steps to generate iteratively the number of clones of each task for each operation mode. Suppose for each task T_i^j the number c_i^j of clones has been determined and we want to generate c_{i-1}^j. First, we choose a random number uniformly distributed in [1, n] and want to delete this number of clones from $RPTS_i$. Second, for each task T_i^j if $c_i^j > i-1$, then $c_{i-1}^j = i-1$. T_{i-1}^j has one clone less than T_i^j. If the total number of clones deleted in this way is more than or equal to the random number chosen in the first step, then we stop and c_{i-1}^j has been determined. Otherwise, let d be the difference. Third, we select d tasks uniformly from $RPTS_i$ and delete one clone from each. A task can be selected multiple times. If a selected task has zero clone, then no clone is deleted. Initially, $RPTS_m$ has an average of $n(m+1)/2m$ clones per PE. We remove $(n+1)/2$ clones on the average from $RPTS_i$ to get $RPTS_{i-1}$. As the number of non–faulty PE's is reduced to 1, the average number of clones per PE is increased to $n-(m-1)/2$. Thus, we simulate the unwillingness to delete the clones from a task system and the increase in the hard workload, as more PE's fail.

The sample size at each data point is 20. The number (m) of non–faulty PE's is set to 20 initially for each simulation. To completely evaluate the allocation algorithm for a replicated imprecise task allocation problem, we need to execute the allocate algorithm m times to solve m sub–problems. Hence, 400 sub–problems are evaluated at each data point. On the other hand, to evaluate the modify algorithm completely, we need to run the allocate algorithm one time and the modify algorithm $m! \sum_{i=1}^{m-1} 1/i!$ times, one for each permutation of 1 to $m-1$ PE failures. The reason is that the order in which the PE's become faulty affects how the modify algorithm changes the existing task allocation and thus the TWU_i produced. When m is a large integer, the complete evaluation of the modify algorithm becomes computationally intractable. Therefore, we evaluate the modify algorithm partially in each operation mode. In mode i, one of i PE's is selected randomly to become faulty. The probability of failure is uniformly distributed over the i non–faulty PE's. Hence, 400 sub–problems are also evaluated at each data point.

The performance of the allocate algorithm and the modify algorithm is measured by two metrics. The first metric is the average ratio $\theta = TWU_i / TWU_i^{upper}$. θ is averaged over the feasible sub–problems at each data point. The upper bound value is used instead of the optimal value because the problem is NP–hard. The ratio θ is a conservative measurement in the sense that TWU_i^{upper} is always greater than the optimal value, when the scheduling algorithm is EDF. When the scheduling algorithm is RM, the ratio θ is not conservative because the schedulability threshold associated with RM is a sufficient (not necessary) bound. The second metric is the number of infeasible task allocations generated at each data point. As discussed earlier, the allocate algorithm and the modify algorithm are heuristic in nature and may generate infeasible task allocations for feasible sub–problems. This metric shows how good the algorithm is in finding feasible solutions. When the scheduling algorithm is EDF, this metric is also conservative in the following sense. For some task systems with high hard workloads, there are no feasible task allocations. However, the greedy algorithm solves the relaxed problem in which two constraints for feasible task allocations are ignored. It may falsely declare some sub–problems feasible that are in fact infeasible.

4.6.2.1. Effect of the Number of Tasks

We are interested in knowing how large n must be to make θ close to 100%. As we increase the number of tasks in a task system, the hard workload of the task system increases. To ensure that the task systems generated are not biased against large numbers of tasks and that few infeasible task systems are generated, we keep the normalized hard workloads ($n\bar{c}_m \bar{u}_h / m$) of the task systems constant and less than 1. This can be done by setting the average hard workload ($\bar{u}_h$) of a task equal to $1/n$. Hence, the range of the utilization (u_m^j) of a task T_m^j is (0 $4/n$). In this simulation, the average normalized hard workload is fixed at 0.5. (More precisely, $\bar{c}_m / m = 0.5125$.) Figure 4.4 shows that θ's approach to 100% monotonically, as the number of tasks increases and the average hard workload ($\bar{u}_h$) decreases. When the number of tasks is 60

and the modify algorithm is used, $\theta(EDF)$ and $\theta(RM)$ are 99.43% and 98.36% respectively. If the allocate algorithm is used instead, $\theta(EDF)$ and $\theta(RM)$ are 99.93% and 99.68% respectively. This result is implied indirectly by Theorem 4.2. The maximum difference in θ between the modify algorithm and the allocate algorithm for each data point is less than 3.5%.

The second and third columns of Table 4.8 list the numbers of infeasible sub–problems found by the greedy algorithm, when the scheduling algorithm is EDF. Only 39 out of 2400 sub–problems are infeasible. The numbers of infeasible task allocations are displayed in the fourth and fifth columns (resp., the sixth and seventh columns), when the scheduling algorithm is EDF (resp., RM). In EDF case, 4 out of 4722 task allocations are infeasible. In RM case, there are more infeasible task allocations because the available processor utilization is lower than that in EDF case. For the same reason, the number of infeasible sub–problems is at least 39, when the number of tasks is 10.

Table 4.8: The number of infeasible sub–problems and the number of infeasible task allocations generated by the algorithms, when the number of tasks varies

Number of Tasks	Infeasible Sub–problems		Infeasible Task Allocations			
	EDF		EDF		RM	
	Allocate	*Modify*	*Allocate*	*Modify*	*Allocate*	*Modify*
10	39	39	1	1	68	85
20	0	0	0	2	71	77
40	0	0	0	0	56	64
60	0	0	0	0	73	83
80	0	0	0	0	78	88
100	0	0	0	0	43	52

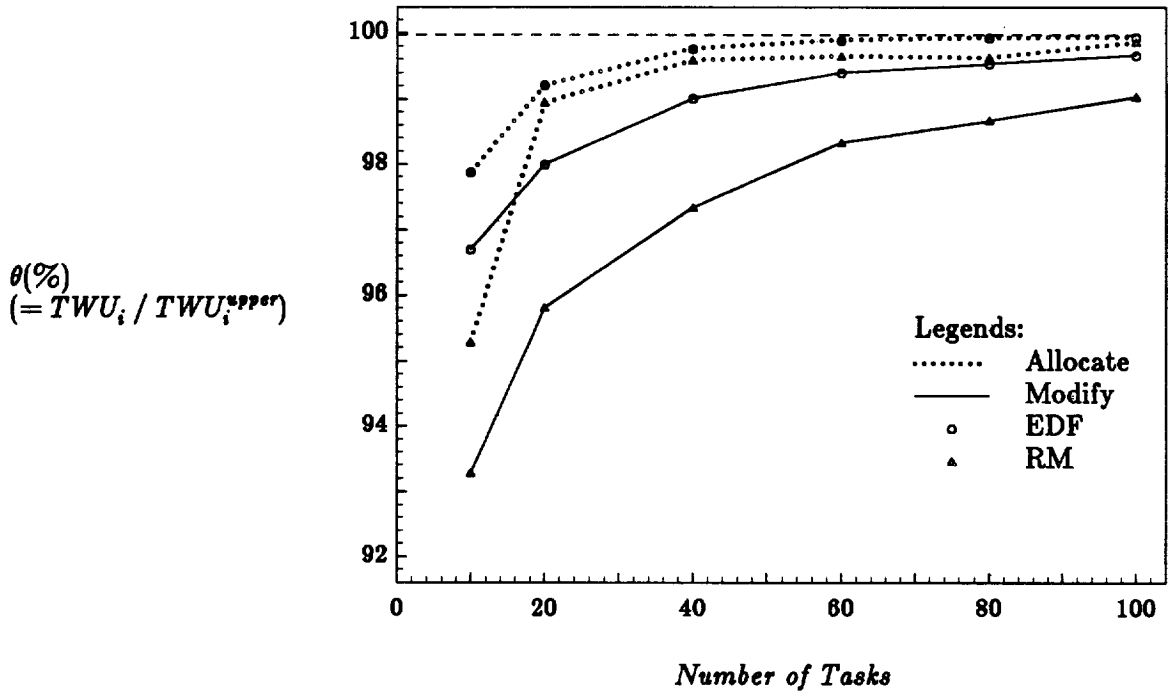


Figure 4.4: θ increases as the number of tasks increases

4.6.2.2. Effect of Hard Tasks

It is more difficult to find a feasible task allocation for a task system with a higher normalized hard workload. In this simulation, we study its effect on θ and the number of infeasible task allocations. We let the number of tasks be 60 and increase the normalized hard workload from 0.375 to 1. When the scheduling algorithm is EDF, the second and third columns of Table 4.9 show that the number of infeasible sub–problems found by the greedy algorithm increases monotonically. Although the fourth and fifth columns do not show monotonic increasing of infeasible task allocations, the column–wise sums do, e.g., the second and fourth columns for the allocate algorithm and the third and fifth columns for the modify algorithm. Moreover, the allocate algorithm produces only 4 infeasible task allocations out of 1799 task allocations. When the scheduling algorithm is RM, the number of infeasible task allocations found by each algorithm increases

monotonically. Table 4.9 also shows that the modify algorithm produces more infeasible task allocations than the allocation algorithms. If the scheduling algorithm is RM and the normalized hard workload is greater than 0.75, the last two columns of Table 4.9 show that most task allocations are infeasible. Indeed, most sub–problems are infeasible because the normalized hard workloads exceed the schedulability threshold. The number of infeasible sub–problems for each data point is no less than that of the EDF case. These results show that the allocate algorithm and the modify algorithm are usually robust and capable of finding feasible task allocations in spite of increasing normalized hard workload.

Figure 4.5 shows that $\theta(EDF)$ and $\theta(RM)$ decrease monotonically, as the workload of the hard tasks gets higher. The worst case $\theta(allocate)$ is still greater than 97% and $\theta(modify)$ 96%. The maximum difference in θ between the allocate algorithm and the modify algorithm is less than 3%.

Table 4.9: The number of infeasible sub–problems and the number of infeasible task allocations generated by the algorithms, when the normalized hard workload varies

Normalized Hard Workload	Infeasible Sub–problems		Infeasible Task Allocations			
	EDF		EDF		RM	
	Allocate	*Modify*	*Allocate*	*Modify*	*Allocate*	*Modify*
0.375	0	0	0	0	3	3
0.5	0	0	0	0	73	83
0.625	44	44	0	5	210	227
0.75	81	81	1	21	319	337
0.875	169	169	3	40	389	391
1.0	307	307	0	18	400	400

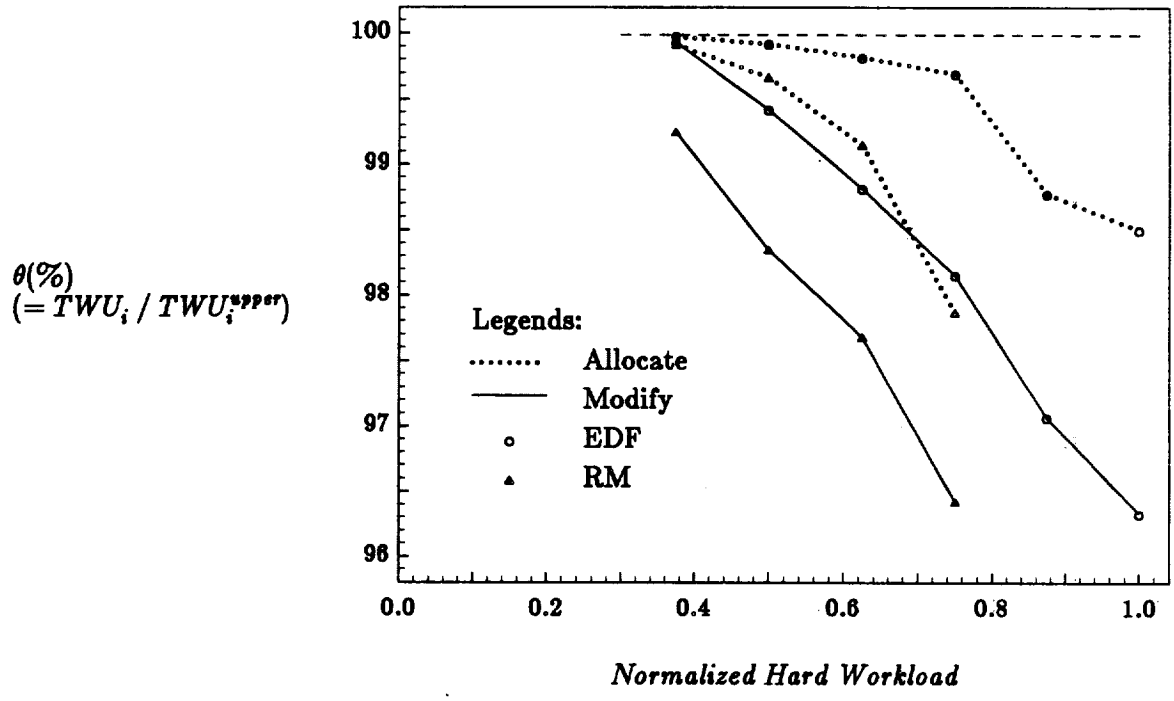


Figure 4.5: θ decreases as the normalized hard workload decreases

4.6.2.3. Effect of Hardware Failures

Finally, we show the effect of hardware component failures. We use the samples in Section 4.6.2.1, and average and present them according to the number of PE failures in stead of the number of tasks. Figure 4.6 shows that the θ curves have shapes similar to a bath tub. As the number of PE failures increases, θ produced by the allocation algorithm has a decreasing trend in the bottom portion of the tub. This is expected because the normalized hard workload increases as more PE's fail. Whereas, θ produced by the modify algorithm does not have a decreasing trend in the bottom portion of the tub. The decreasing trend expected of the modify algorithm may be offset by the additional constraint that the modify algorithm maintains the execution continuation of the existing tasks. The zigzagging curves may be explained by the probabilistic manner by which the clones are deleted from $RPTS_i$. The normalized hard workload of $RPTS_i$

may be greater than that of $RPTS_{i-1}$. The rising tail curve is because TWU_i of infeasible task allocations are excluded from θ. In particular when the number of non–faulty PE is one, task allocations are either infeasible or optimal. The maximum difference in θ between the allocate algorithm and the modify algorithm is less than 3.5%.

As more PE's become faulty, Figure 4.7 shows that when the scheduling algorithm is RM the number of feasible task allocations generated by both algorithms tends to decrease. This is because the normalized hard workload approaches to one, exceeding the RM schedulability threshold. In contrast, when the scheduling algorithm is EDF, the number of feasible task allocations stay almost constant. The normalized hard workload in this case is less than the EDF

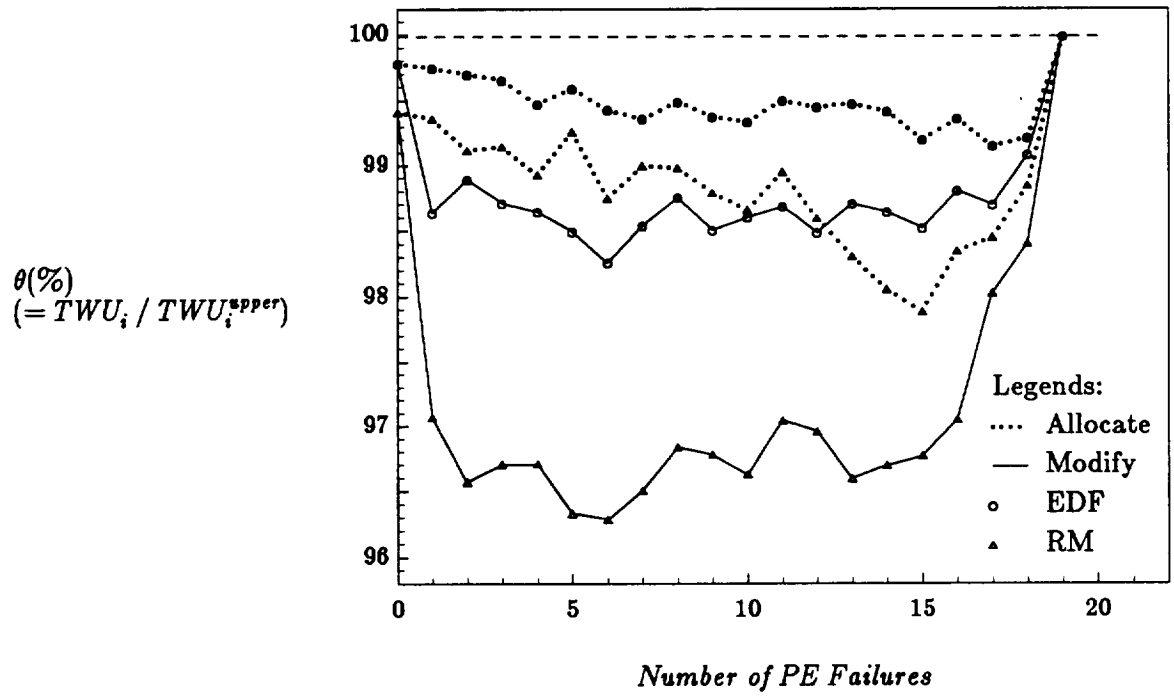


Figure 4.6: The change in θ as more PE's become faulty

schedulability threshold. The allocation algorithm always generates more feasible task allocations out of 120 sub–problems for each data point than the modify algorithm. Yet, the maximum difference is 7.

4.7. Summary

We have presented two algorithms for scheduling real–time replicated tasks under hardware failures. When hardware components fail, less computational resources are available. The

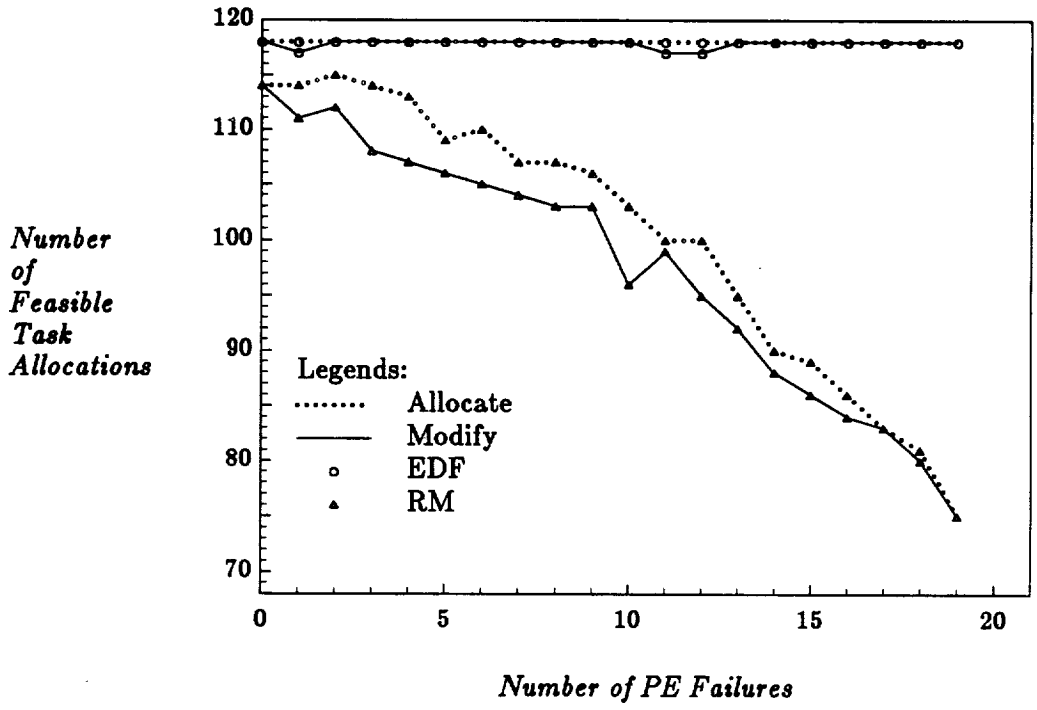


Figure 4.7: The number of feasible task allocations generated by the algorithms as more PE's become faulty

imprecise computation model provides the scheduling algorithms with the flexibility of trading off the quality of the results with the amount of computational resources required to produce them. Furthermore, when less computational resources are available, real–time systems may require a fewer number of clones to perform critical functions. As a result, these two flexibilities permit the performance of the real–time system to degrade gracefully under hardware failures.

The allocate algorithm can be used either before run time or on–line. In the former case, no computational resources are needed to generate task allocations at run time. However, similar to the problem pointed out in the reference [68], the allocate algorithm does not maintain continuity with respect to the previous task allocation. The modify algorithm avoids this problem by not migrating the clones already allocated to non–faulty PE's. Since which PE's will malfunction at run time cannot be known a priori, the modify algorithm is generally executed on–line.

We have evaluated the algorithms by stochastic analysis and simulations. The simulation results show that both algorithms are effective under hardware failures. The allocate algorithm oftentimes generate (almost) optimal solutions, while it seldom produces infeasible task allocations. In almost all cases, the average TWU_i/TWU_i^{upper} produced by the allocate algorithm is only less than 3.5% higher than that produced by the modify algorithm. The analytical results show that when the priority driven scheduling algorithm used is EDF, the LUF algorithm produced task allocations whose maximum processor utilizations approach the optimal asymptotically, as the number of tasks increases. The analytical results also show that the ratio of the worst–case maximum processor utilization and the optimal maximum processor utilization is bounded by a small number.

Chapter 5
Recovery Manager for Replicated Periodic Tasks

In Chapter 4, we assume that the operating system can detect, isolate, and remove failed PE's from service. In this chapter, we present the system supports to enable the O.S. to have these capabilities. Section 5.1 describes a strategy for responsive and imprecise repairs. Section 5.2 discusses briefly the research related to fault–tolerant distributed real–time systems. Section 5.3 and 5.4 describe the assumptions made in the system architecture and fault model in addition to those in Section 4.3. Section 5.5 describes the schemes whereby replicated tasks detect errors. We present the diagnosis algorithm in Section 5.6 and the repair algorithm which allows imprecise results in Section 5.7. A summary can be found in Section 5.8.

5.1. Responsive and Imprecise Repairs

Chapter 4 describes a fault–tolerant software architecture for replicated real–time tasks on multiprocessor systems. The architecture incorporates the redundancy and masking technique and the imprecise computation model. In this architecture, a real–time system has n periodic tasks each of which may be replicated. The copies or *clones* of a task must be assigned to distinct processing elements (*PE*'s), and each clone must receive at least some minimum amount of processing time in order to produce an acceptable result. The number of clones to be allocated to a task is monotonically non–increasing with the number of non–faulty PE's. The clones allocated to a PE are scheduled by the priority–driven scheduling algorithms, i.e., the rate monotonic algorithm (*RM*) or the earliest deadline first algorithm (*EDF*). We have designed two algorithms: The allocation algorithm seeks an initial clone allocation for each task, and, when there is a PE failure, the modify algorithm finds a new clone allocation with the least clone migration from the old configuration.

In hard real–time systems, it is important to detect errors and initiate repairs as soon as possible. The redundancy and masking technique not only hides the PE failures from the applications. It also permits fast error detection for different classes of faults [17]. If a clone executing on a PE finds other PE's (or clone's) to have some unexpected behavior, it reports the suspected faulty PE's to the O.S. The O.S. then initiates the *diagnosis servers* to poll the suspected PE's. If the PE's do not respond to the polls, then the diagnosis servers inform the *repair server* to generate a new clone allocation in which some of the clones assigned to the faulty PE's are re–allocated to the non–faulty PE's, and the clones on the non–faulty PE's may be assigned a shorter processing time.

One of the problems in this architecture is that the diagnosis servers and the repair server may be subject to PE failures themselves. If the repair server is aborted due to its host PE failure, the system may be installed partially with the new clone allocation. Moreover, while the repair server is handling a PE failure, other clones may report the same or additional failures. One solution is to handle these new failures sequentially, and to generate a new re–allocation for each of the failures. However, such sequential processing would mean long delays for multiple failure recoveries and may not be acceptable for real–time applications.

We propose to implement the repair server as an *imprecise computation* so that it can be interrupted any time and behave gracefully. The reason why we can implement the repair server as an imprecise computation is because PE diagnosis reports never contradict with each other. When a PE is faulty, we assume it will not become normal by itself immediately. Moreover, we know that the modify algorithm does not migrate the existing clones from a non–faulty PE to another. It monotonically reduces the number of the excessive clones and the number of the missing clones to zero. Therefore, if we are handling a failed PE when another PE fails, we terminate the ongoing repair service. Any imprecise re–allocation result is in fact an approximation of the precise re–allocation. We then start from the current clone allocation to generate a new clone allocation in which both failed PE's are removed from the system configuration.

To make PE failure recovery even more timely, we propose to implement the diagnosis servers as replicated but independent servers [64]. Each of the diagnosis servers will be responsible for monitoring some number of PE's. Each PE, on the other hand, is monitored by more than one diagnosis server. In this way, if one server is busy polling a suspicious PE, we can find another server to poll another suspicious PE. Moreover, if a diagnosis server fails, we can always find other diagnosis servers to cover its monitoring PE set.

5.2. Related Research

In Mars, time redundancy and shadow PE's are introduced to tolerate transient and permanent hardware faults [40, 41]. In addition to extensive checks in the hardware and in the O.S., time–redundant execution of application tasks is proposed for detecting transient faults. The average processing time of a real–time task is shorter than its worst–case processing time. To capitalize this difference, a task is given a processing time which is sufficiently long to assure 99% certainty that the second execution of the task on the same PE will complete. If this processing time is shorter than the worst–case processing time, the task is given the worst–case processing time.

Similar to TMR (triple–module–redundancy), a shadow PE operates in redundancy with two other actively redundant PE's. When a transient fault is detected, the affected PE is turned off and re–integrated immediately by retrieving the uncorrupted state of the actively redundant partner PE. When a permanent fault is detected, the shadow PE assumes the role of the affected PE. They showed that shadow PE's reduce the probability of spare exhaustion. However, if a task is not critical, its degree of replication remains fixed and none of its clones is re–allocated.

Distributed execution of recovery blocks was proposed as an uniform approach to tolerating permanent and transient hardware faults, and software faults [38, 39]. Two schemes based on this concept were formulated and experimented. The experimental results show that both schemes have small execution overheads in term of time costs. The first scheme exploits the

concurrent execution of try blocks to facilitate fast forward recovery. When PE's are fault–free, the primary and shadow PE's will pass the acceptance test with the results computed with their primary try blocks. If the primary PE fails and the shadow PE passes its test, the shadow PE assumes the role of the primary, i.e., the PE's exchange their roles. On the other hand, if the shadow PE fails first, the primary PE need not be disturbed. In both cases, the failed PE attempts to become an operational shadow PE; it attempts to roll back and retry with its alternative try block to bring its application computation state or local database up–to–date. This attempt does not disturb the primary PE.

For the applications where rollback–and–retry is an acceptable mode of recovery, the second scheme is more cost–effective. The second scheme allows each PE to dynamically select its next task among the tasks of executing the primary try block, the alternate try block, and the acceptance test. Hence, the load balancing technique used in the second scheme is self–scheduling. If the result produced by the primary try block does not pass the acceptance test, the alternative try block is executed next. The drawback of this scheme is, as pointed out in the paper [39], that an "insane" PE can significantly disrupt the system. Therefore, PE's should have only crash failure semantics.

5.3. System Architecture

Figure 5.1 shows the levels of abstraction in the systems. The processor–level architecture is composed of m PE's interconnected by a communication network. Each PE has only the crash/timing failure semantics. It is the so–called "fail–stop" (fail–fast, fail–silent) PE for the non–time–critical applications. It satisfies the halt on failure property and the failure status property [41, 71, 72]. A fail–stop PE can be implemented by multiple tightly–coupled lock–step components (CPU's, message processing units (MPU's), and busses) and the error detection self–checking circuits [40, 70, 92]. The error detection and correction code is used in the memories. These fault–tolerant techniques transform malicious failures to crash/timing failures.

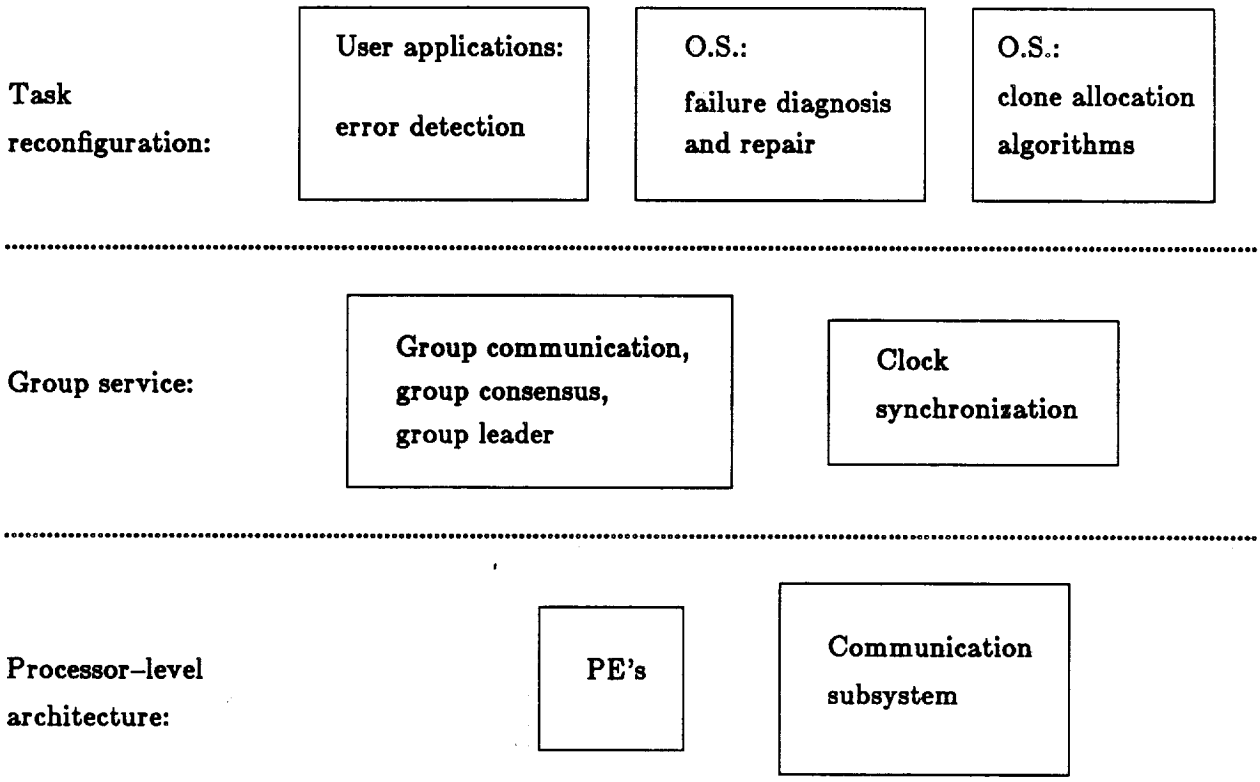


Figure 5.1: Levels of abstraction in the systems

The communication network is assumed to be reliable and fault–tolerant. It supports reliable broadcasts [8, 40, 86]. Every non–faulty PE receives the broadcast message. The error detection and correction code is used in the messages. The receiver of a message knows who sent the message. Every PE appends its own signature to every message it sends and no PE can forge the signature of any other non–faulty PE. The absence of a message can be detected. MPU's time–stamp every incoming message and every outgoing message [40]. The clocks of all non–faulty PE's are synchronized [69]. The time difference in any two clocks is bounded from above by a constant.

The O.S. services are provided by a group of sporadic or aperiodic servers [82]. There are two opposing approaches to implementing reliable O.S. services [17]. In the *active* redundancy

approach, several servers can simultaneously provide services. In contrast, only one active server can provide services in the *standby* redundancy approach. If the active server fails, one of the standby servers will take over its job. The active redundancy approach provides faster services than the standby redundancy approach and is adopted for real–time tasks in user applications. For diagnosis services, we choose a hybrid approach in which the diagnosis servers are independently replicated to provide concurrent, independent services and the standby redundancy approach is used for coordinating the diagnosis servers. For repair services, we need to ensure that the system configuration is uniquely defined and therefore choose the standby redundancy approach. The active repair server decides the new allocation, but can be interrupted any time when there is a new PE failure. Each interruption will produce a partial reconfiguration which can be used to keep some of the critical jobs to continue their minimum operations.

5.4. Fault Model

The fault model in this study includes transient and permanent physical faults in the PE's. Transient faults may be caused by lightening, electric–magnetic radiation, alpha particles, cosmic rays, high operational and environment temperature, short deviations of the supply voltage, etc. Design faults are not considered in this paper.

We assume that transient faults are not likely to occur simultaneously to all clones of a task, if the number of clones is sufficiently large. The allocate algorithm and the modify algorithm tend to assign different sets of clones to distinct PE's. The execution intervals produced by EDF or RM for the clones of a periodic task may not be identical.

We assume that permanent faults can be uncovered by testing [47], while transient faults cannot. If a suspicious PE passes the test, it has transient faults and can be restarted [41]. Moreover, the mean time between failures is much greater than the time needed for a failure recovery. If more PE's fail during a failure recovery, the system will remain functional as long as there are enough existing clones to mask the additional failures. Our repair algorithm is capable

of fixing multiple PE failures at a time.

5.5. Detection of Errors

In the replicated imprecise task allocation problem described in Section 4.3, replicated tasks use a replication resolution scheme for masking crash/omission/timing failures. This scheme assumes that the same amount of processing time is assigned to the clones of a task. The non–faulty clones produce the same result. During a clone reconfiguration, different amounts of processing time may be assigned to the clones of a task. The non–faulty clones may produce different and yet acceptable results. Another scheme is needed for this case and is described next.

Suppose there are two periodic tasks T^1 and T^2. The result produced by T^1 is consumed by T^2. When T^1 and T^2 are replicated, the result produced by each clone of T^1 is broadcast to all clones of T^2. The additional processing time needed for broadcasting is added to the processing time of T^1. In this resolution scheme, each clone of T^2 determines the best result by the error indicators associated with the imprecise results [59]. Each clone of T^2 chooses the best result among those that arrive on time as its input. The additional processing time needed for choosing the best result is added to the processing time of T^2. Again, neither agreement nor synchronization among the clones of T^2 is needed. If T^1 has c clones, T^1 can tolerate up to $c-1$ crash/omission/timing failures [17].

To detect crash/omission/timing failures, we need to find out those clones of T^1 that produce late or no results. In particular, the results have to be time–stamped for detecting timing failures. Although a clone of T^2 is ready for execution at its ready time, it may not be executed until later. So, the late results that arrive before the execution of T^2 may not be detected without time stamps. The time needed for detecting crash/omission/timing failures is $O(c)$. A clone of T^2 checks the time stamps of the results produced by all clones of T^1 sequentially to see if they are late or missing.

The error latency between a fault occurrence and its detection is bounded from above by $p^1 + p^2$ where p^1 and p^2 are the periods of T^1 and T^2 respectively. Suppose a fault occurs during a period of T^1 and on the PE_1 to which a clone of T^1 is allocated. If a clone of T^2 runs on a non-faulty PE, it will detect the erroneous result produced by the clone of T^1 during a period of T^2 immediately following the period of T^1. The detection of the faulty PE_1 is no later than the end of the period of T^2. Hence, the maximum latency for detecting the failure of T^1 is $p^1 + p^2$.

When a clone of T^2 uncovers the PE failures, it sends a record of four fields to the O.S. of its host PE. The first and second fields are the task ID's of T^1 and T^2 respectively. The third field is the time when the failure is detected. The fourth field is a list of tuples each of which has the ID of a failed PE and the corresponding class of failures. The O.S. then tries to send the record to the coordinator (to be explained in the next section) until it receives a positive acknowledgement. If the host PE fails before the record is delivered, the record is lost.

5.6. Diagnosis

When a PE malfunctions, several clones may independently report the same PE failure to different diagnosis servers. If the diagnosis of the suspected faulty PE is not coordinated, these diagnosis servers may poll the same PE concurrently. Although active redundancy provides a faster service, it creates additional computational loads which may in turn delay the diagnosis of other suspicious PE's. Moreover, the uncoordinated diagnosis servers may interrupt the repair server multiple times for the same PE failure. Therefore, it is important to coordinate the diagnosis servers so that they independently investigate distinct PE failures. In this way, the overall diagnosis of the suspicious PE's is faster.

We solve this problem by a divide and conquer strategy. The PE's in a real-time system are partitioned into clusters [40, 64]. Each cluster has a coordinator and several diagnosis servers. In a cluster, the function of the coordinator is to collect the failure reports of the PE's in the cluster, coordinate the diagnosis servers, and interrupt the repair server only once for each

PE failure. The number of PE's and the number diagnosis servers in a cluster are engineering parameters. They are determined by the system requirements and architecture.

5.6.1. Coordinator

The coordinator of a cluster has an aperiodic task server accepting two types of requests. First, the diagnosis–coordinator algorithm accepts the reports of the failed PE's in the cluster and initiates the diagnosis of these PE's (Figure 5.2). If the suspected faulty PE's in the reports are already in the faulty PE set or being diagnosed, the reports are discarded. S and F_c are the sets of suspicious and faulty, respectively, PE's in the cluster. R is the set of PE's for which outstanding diagnosis requests have been issued to the diagnosis servers by the coordinator in the cluster. Otherwise, for each suspicious PE not in F_c nor in R, the coordinator selects a diagnosis server ($diagnosis-server_{min}$) in the cluster and sends a request to the server. The selection of $diagnosis-server_{min}$ depends on the loads of the diagnosis servers in the cluster and the communication costs between the suspicious PE and the diagnosis servers. A simple heuristic rule for load balancing is as follows. If several diagnosis servers are idle, the coordinator sends the request to the server with the least communication cost. Otherwise, the coordinator sends the request to the server with the fewest requests.

Second, the repair–coordinator algorithm accepts the diagnosis results from the diagnosis servers and initiates a restart or a clone reconfiguration (Figure 5.3). When a diagnosis server completes the testing of a suspicious PE, it informs the coordinator of the test result. If the result shows a permanent failure, the PE is added to F_c. The coordinator starts a clone reconfiguration by broadcasting the new faulty PE to the active and standby repair servers. If the result indicates a transient failure, the coordinator tells the repair servers to restart the PE. The failure status of the PE is changed to non–faulty.

We assume the time for a diagnosis is longer than the maximum error latency of all periodic tasks. Otherwise, the coordinator records the earliest time when the failure of a PE is first

```
Algorithm diagnosis–coordinator
input: S;
begin
        for each PE ∈ S do
                if (PE ∉ (F_c ⋃ R)) then
                begin
                        diagnosis−server_min = load–balance();
                        send(diagnosis−server_min, PE);
                        R = R ⋃ PE;
                end;
end;
```

Figure 5.2: The diagnosis–coordinator algorithm

```
Algorithm repair–coordinator
input: PE, status;
begin
        if (status = OK) then
                broadcast(repair−servers, restart, PE);
        else
        begin
                F_c = F_c ⋃ PE;
                broadcast(repair−servers, clone–reconfiguration, PE);
        end;
        R = R − PE;
end;
```

Figure 5.3: The repair–coordinator algorithm

detected. The earliest time must be after the ready time of the period of T^2 during which the PE failure is detected. The coordinator dismisses all the failure reports of the same PE in which the failure detection times (the third fields of the reports) are before the earliest detection time plus the maximum latency.

5.6.2. Diagnosis Servers

A diagnosis server accepts the requests from the coordinator in a cluster and processes the requests serially (Figure 5.4). It is an aperiodic task server. The diagnosis server polls the suspicious PE in a request [47]. If the PE passes the poll, it has transient faults. Otherwise, it has permanent faults. The diagnosis server reports the outcome of the poll to the coordinator.

5.6.3. Redundancy and Consistency

The coordinator in a cluster has standby redundancies which can be described by a directed *chain* in which the $(i+1)^{th}$ element is the standby task server of the i^{th} element. The first element of the chain is the active server. The $(i+1)^{th}$ element monitors the i^{th} element by polling the host PE of the i^{th} element periodically to see if the PE has failed stop. Hence, the standby servers have periodic tasks. The elements of a chain are assigned to distinct PE's. If the host PE of the i^{th} element has failed, the $(i+1)^{th}$ element informs the coordinator of the PE failure and assumes the role of the i^{th} element. The number of standby servers in the chain is determined by the reliability requirements of the system. The length of the monitoring period is an engineering parameter. While a shorter period gives faster detections of PE failures, a longer period has less monitoring overheads.

```
Algorithm diagnosis
input: PE;
begin
        status = poll(PE);
        send(repair−coordinator, PE, status);
end;
```

Figure 5.4: The diagnosis algorithm

When the host PE of the active coordinator server fails, the second element in the chain becomes the new active server. If the PE failure is transient, when the PE is restarted it becomes the last element of the chain. If the PE failure is permanent, a standby server may be added to a non–faulty PE in the cluster during the clone reconfiguration. In both cases, if there are suspicious PE's in S, the new active server tries to complete the diagnosis of these suspicious PE's.

In each cluster, the active coordinator monitors the diagnosis servers periodically. Hence, it also has a periodic task. When the coordinator detects the host PE failure of a diagnosis server, it requests another diagnosis server to poll the failed host PE. If the failed diagnosis server has incomplete requests, the coordinator distributes these requests to the other diagnosis servers. If the host PE failure is permanent, a diagnosis server may be added to a non–faulty PE in the cluster during the clone reconfiguration.

The diagnosis state of a cluster is characterized by the values of S, F_c, and R in its coordinator. A state is *consistent* if F_c and R do not intersect. Since both the diagnosis–coordinator algorithm and the repair–coordinator algorithm access the shared variables F_c and R, we make the execution of both algorithms atomic to maintain a consistent state [54]. Each atomic execution of the diagnosis–coordinator algorithm or the repair–coordinator algorithm moves the state of the coordinator from a consistent state to another. Whenever the active server changes its state, it broadcasts S, F_c, and R to the standby servers to ensure that the non–faulty ones make the same state transition.

Example

The real–time system in the example has only one cluster consisting of seven PE's. Table 5.1 lists the tasks in the system. The task parameters are explained in Chapter 4. For example, the period (p) of T^1 is 20. The complete processing time (τ) of T^1 is 4. The processing time (h) of its hard task is 2. Its weight (w) is 7. When no PE fails, T^1 has 3 clones (c_7). When two PE's fail, T^1 has 2 clones (c_5). The clones allocated to a PE are scheduled according to the EDF

priority–driven scheduling algorithm.

When the system has no PE failure, it has one active and three standby coordinators represented by the clones of T^2, one active and three standby repair servers represented by the clones of T^3, and four diagnosis servers represented by the clones of T^{10}. We use the period transformation method to shorten the periods of the coordinators and repair servers so that they have faster response times [74]. The diagnosis servers have a longer period to keep the number of preemptions small. We assign the same processing time to the standby servers as the active server. When the active server fails and one of the standby severs becomes active, we want the standby server to have the same performance as the active server. Table 5.2 shows the initial clone allocation generated by the allocate algorithm for the system.

PE_3 and PE_5 malfunction due to some external physical disturbances. When the active coordinator receives the failure reports of PE_3 and PE_5, it treats them as suspected faulty PE's. $S = \{PE_3, PE_5\}$. The active coordinator sends two diagnosis requests, one for PE_3 and the other

Table 5.1: The periodic tasks in the system

j	p	τ	h	w	c_7	c_6	c_5	c_4	c_3	c_2	c_1
1	20	4	2	7	3	3	2	2	1	1	0
2	30	2	1	3	4	4	3	3	2	2	1
3	30	4	2	5	4	4	3	3	2	2	1
4	40	12	8	10	3	2	2	2	1	1	0
5	50	23	11	9	4	4	3	3	2	2	1
6	100	29	24	4	4	4	4	3	3	2	1
7	100	41	28	2	2	2	1	1	0	0	0
8	200	90	37	8	3	3	2	1	1	0	0
9	300	78	49	1	2	1	1	1	0	0	0
10	300	33	15	6	4	4	3	3	2	2	1

Table 5.2: The initial allocation of the clones in the system

PE	j 1	2	3	4	5	6	7	8	9	10	Proc. Util.
1	0	1	1	0	0	0	1	1	1	0	1
2	1	0	1	0	0	1	0	1	0	1	1
3	1	1	1	0	0	1	0	1	0	0	0.987
4	0	1	0	1	1	1	0	0	0	0	0.997
5	1	1	0	1	1	0	0	0	0	1	1
6	0	0	1	1	1	0	0	0	1	1	1
7	0	0	0	0	1	1	1	0	0	1	0.993

for PE_5, to the diagnosis servers on PE_2 and PE_6. $S = \emptyset$ and $R = \{PE_3, PE_5\}$. When the diagnosis servers complete the tests, the diagnosis results show that both PE_3 and PE_5 have permanent faults. The diagnosis servers pass the results to the coordinator in the cluster. $R = \emptyset$ and $F_c = \{PE_3, PE_5\}$. The coordinator broadcasts PE_3 and PE_5 to the active and standby repair servers.

5.7. Repair

A repair is either a clone reconfiguration or a PE restart. Since the coordinators send repair requests to the repair server asynchronously, the interferences among clone reconfigurations and PE restarts can be complex. The repair server must process the requests timely and robustly so that the system configuration is uniquely defined and the execution states of the clones of a task are aligned. We describe first the algorithm for a clone reconfiguration and the algorithm for a PE restart as though there is no interference. Then we describe how the repair server resolves the interferences.

C-2.

5.7.1. Clone Reconfiguration

A clone reconfiguration (Figure 5.5) has two phases: (1) the generation of a new clone allocation, and (2) the installation of the new clone allocation on the non–faulty PE's. In the first phase, the repair server constructs the new clone allocation $\{new-x_i^j\}$ by calling the modify algorithm with the new faulty PE set ($new-F_r$) and the current clone allocation $\{x_i^j\}$. x_i^j is the processing time assigned to the clone of task T^j allocated to PE_i. If x_i^j is zero, then a clone of task T^j is not allocated to PE_i.

In the second phase, the clone–reconfiguration algorithm sends a request to each non–faulty PE_i to decrease or increase the processing times of the clones on PE_i. N denotes a sorted list of

```
Algorithm clone–reconfiguration
input: new−F_r;
begin
    {new−x_i^j} = modify(new−F_r, {x_i^j});          <phase 1>

    for each PE_i ∈ N do                              <phase 2>
    begin
        send(PE_i, {new−x_i^j});
        if (positive acknowledgement) then
        begin
            {x_i^j} = {new−x_i^j};  <vector assignment.>
            broadcast(repair−standby−servers, {x_i^j});
        end;
        else
            send(diagnosis−coordinator, PE_i);
    end;

    F_r = new−F_r;
    broadcast(standby−repair−servers, F_r);
end;
```

Figure 5.5: The clone–reconfiguration algorithm

the non–faulty PE's which are the complement of $new-F_r$ with respect to all PE's. N is sorted in the increasing order of PE ID. If the outcome of the send to PE_i is a positive acknowledgement, the active repair server broadcasts the update, $\{x_i^j\}=\{new-x_i^j\}$, to the standby repair servers. The broadcast marks the transition from the current clone allocation to the next. Otherwise, the repair server sends a request to the coordinator in the host cluster of PE_i to determine if PE_i is faulty. If so, the coordinator starts another clone reconfiguration. A clone reconfiguration is completed, when the active repair server broadcasts the new faulty set to the standby repair servers.

After PE_i receives the new processing times, it calls the decrease–increase algorithm (Figure 5.6) to first reduce the processing times of those clones whose new processing times are smaller. Then, it increases the processing times of those clones whose new processing times are larger. When the processing time of a clone is reduced to zero, the clone is deleted from the task queue of PE_i. When the processing time of a clone is increased from zero to a non–zero value, the clone is added to the task queue of PE_i.

When the processing times of clones are changed, some care is needed to avoid undesirable transient effects [75]. Since the clone reconfiguration algorithm does not change the processing times of the clones of a task simultaneously, we need to align the execution states of these clones so that they are identical. The algorithm for clone alignment can be found in Section 5.7.4.

The time complexity of the modify algorithm is $O(n^3)$ as shown in Chapter 4. The clone–reconfiguration algorithm sends out $O(m)$ messages whose length is $O(n)$. The time complexity of the decrease–increase algorithm is $O(n)$. The time complexity of the clone–reconfiguration algorithm is $O(n^3+mn)$.

5.7.2. PE Restart

When the repair server restarts a PE, say PE_i, the repair server sends PE_i its current clone allocation $\{x_i^j\}$. PE_i runs the restart algorithm (Figure 5.7) to install its current clone allocation

```
Algorithm decrease–increase
input: {new−x_i^j};
begin
        for each new−x_i^j ∈ {new−x_i^j} do
                if (new−x_i^j < local−x_i^j) then
                begin
                        local−x_i^j = new−x_i^j;
                        align(C_i^j)
                end;

        for each new−x_i^j ∈ {new−x_i^j} do
                if (new−x_i^j > local−x_i^j) then
                begin
                        local−x_i^j = new−x_i^j;
                        align(C_i^j)
                end;
end;
```

Figure 5.6: The decrease–increase algorithm

received from the repair server. The execution state of each clone needs to be aligned so that its state is identical to that of the existing clones of the same task. The align algorithm is presented in Section 5.7.4.

5.7.3. Interferences

The repair server must resolve three kinds of interferences: reconfiguration–reconfiguration, restart–reconfiguration, and reconfiguration–restart. A PE restart does not interfere with another PE restart. The repair sever processes PE restarts serially.

5.7.3.1. Reconfiguration–Reconfiguration Interference

When the coordinators find new permanent PE failures, they interrupt the repair server for clone reconfigurations. If an ongoing clone reconfiguration is interrupted, the server aborts this reconfiguration and starts a new reconfiguration. Section 5.7.6 shows that an incomplete clone

```
Algorithm restart
input: {x_i^j};
begin
        for each x_i^j ∈ {x_i^j} do
        begin
                local−x_i^j = x_i^j;
                align(C_i^j)
        end;
end;
```

Figure 5.7: The restart algorithm

reconfiguration is indeed an imprecise computation.

5.7.3.2. Reconfiguration–Restart Interference

When the coordinators find new transient PE failures, they interrupt the repair server for PE restarts. If an ongoing clone reconfiguration is interrupted, the repair sever tries to combine the PE restarts with the calls to the decrease–increase algorithm in the second phase of the clone–reconfiguration algorithm. The second phase is indeed a sequence of restarts for all non–faulty PE's. The decrease–increase algorithm is more general than the restart algorithm. When the processing time of each clone is zero initially (i.e., $local{-}x_i^j{=}0$) as in the case of a PE restart, the decrease–increase algorithm is no different from the restart algorithm. Specifically, the repair server reconfigures each non–faulty PE sequentially in increasing order of PE ID. (N is a sorted list of the non–faulty PE's.) If a coordinator interrupts the repair server for a PE restart before the server sends this PE the current clone allocation during the second phase, the server simply ignores this interrupt. Otherwise, the PE restart interrupt arrives too late. The repair server queues the PE restart request and processes all queued restart requests after the ongoing clone reconfiguration is completed. Specifically, the clone–reconfiguration algorithm has found that the PE failed stop and has notified the coordinator about the failure. If the coordinator finds the PE either is being diagnosed or has just been diagnosed (due to the error latency), the failure report

is dismissed.

5.7.3.3. Restart–Reconfiguration Interference

When the coordinators find new permanent PE failures, they interrupt the repair server for clone reconfigurations. If an ongoing PE restart is interrupted, the sever discards this and all pending restarts.

Example

We continue with the previous example to illustrate a reconfiguration–reconfiguration interference and a reconfiguration–restart interference. While the active repair server is generating a new clone allocation which removes PE_3 and PE_5 from the system configuration ($new\text{-}F_r = \{PE_3, PE_5\}$), more faults occur on PE_4 and PE_7. The coordinator is notified about the new PE failures and requests the diagnosis servers on PE_2 and PE_6 to poll PE_4 and PE_7. $R = \{PE_4, PE_7\}$. The diagnosis servers find that PE_4 has transient faults and PE_7 has permanent faults, and send the diagnosis results to the coordinator. $R = \emptyset$ and $F_c = \{PE_3, PE_5, PE_7\}$.

Meanwhile, the repair server has generated a new clone allocation and is reconfiguring the clones in the system as shown in Table 5.3. Each $1 \rightarrow 0$ in the clone allocation matrix denotes either the failure of a clone or the deletion of an excessive clone. Each $0 \rightarrow 1$ denotes the addition of a missing clone. In the processor utilization column, $0.997 \rightarrow 1$, for example, means that the utilization of PE_4 is increased from 0.997 to 1.

The coordinator interrupts the repair server for another clone reconfiguration and a PE restart before the repair server installs the new clone allocation on PE_4 and after it installs the new clone allocation on PE_2. $new\text{-}F_r = \{PE_3, PE_5, PE_7\}$. The ongoing clone reconfiguration is aborted. The clone allocation produced by the incomplete clone reconfiguration at the moment of the interrupt is displayed in Table 5.4.

Table 5.3: The clone allocation matrix and the processor utilizations for the first clone reconfiguration

PE	1	2	3	4	*j* 5	6	7	8	9	10	*Proc. Util.*
1	0	1	1	0	0	0→1	1→0	1	1	0	1→1
2	1	0	1	0	0	1	0	1	0	1	1→1
3	1→0	1→0	1→0	0	0	1→0	0	1→0	0	0	0.987→0
4	0	1	0	1	1	1	0	0	0	0	0.997→1
5	1→0	1→0	0	1→0	1→0	0	0	0	0	1→0	1→0
6	0→1	0→1	1	1	1	0	0	0	1→0	1	1→1
7	0	0	0	0	1	1	1	0	0	1	0.993→1

Table 5.4: The clone allocation matrix and the processor utilizations of the incomplete clone reconfiguration

PE	1	2	3	4	*j* 5	6	7	8	9	10	*Processor Utilization*
1	0	1	1	0	0	1	0	1	1	0	1
2	1	0	1	0	0	1	0	1	0	1	1
3	0	0	0	0	0	0	0	0	0	0	0
4	0	1	0	1	1	1	0	0	0	0	0.997
5	0	0	0	0	0	0	0	0	0	0	0
6	0	0	1	1	1	0	0	0	1	1	1
7	0	0	0	0	1	1	1	0	0	1	0.993

The repair server generates a new clone allocation from the current clone allocation. Since the restart request for PE_4 arrives before the the second clone reconfiguration begins, the request is omitted. The restart request is performed equivalently by the decrease–increase algorithm called in the second phase of the clone–reconfiguration algorithm. Table 5.5 shows the clone allocation matrix and the processor utilizations produced by the modify algorithm for the second

clone reconfiguration.

In the old clone reconfiguration (Table 5.3), two clones are added to and one clone is removed from PE_6. In the new clone reconfiguration (Table 5.5), these clones are removed from and added to PE_1 and PE_2. No clone is allocated to or removed from PE_6. Only the processing times of some existing clones on PE_6 are changed.

Table 5.6 shows the processing time and the corresponding utilization assigned to each task before and after each clone reconfiguration. Although the processing time of T^2 is reduced, the coordinator may not necessarily have a longer response time. The expected workload of the coordinator has also decreased because of fewers clones and PE's.

5.7.4. Alignment of the Lost–Soul Clones

The clones of a task are said to be aligned if they have the same execution state. In other words, they produce the same output if given the same input. When restarting a PE which has

Table 5.5: The clone allocation matrix and the processor utilizations for the second clone reconfiguration

					j						
PE	1	2	3	4	5	6	7	8	9	10	*Proc. Util.*
1	0→1	1	1	0	0	1	0	1	1→0	0→1	1→1
2	1	0→1	1	0	0→1	1	0	1→0	0	1	1→0.77
3	0	0	0	0	0	0	0	0	0	0	0
4	0	1	0	1	1	1	0→1	0	0	0	0.997→1
5	0	0	0	0	0	0	0	0	0	0	0
6	0	0	1	1	1	0	0	0	1	1	1→0.823
7	0	0	0	0	1→0	1→0	1→0	0	0	1→0	0.993→0

Table 5.6: The processing time and the corresponding utilization of each task before and after each clone reconfiguration

	Before		Intermediate		After	
j	x_7	u_7	x_5	u_5	x_4	u_4
1	3.867	0.193333	3.167	0.158333	3.2	0.160000
2	1.1	0.036667	1	0.033333	1	0.033333
3	2	0.066667	2	0.066667	2	0.066667
4	12	0.300000	12	0.300000	9.067	0.226667
5	21	0.420000	19.5833	0.391667	11	0.220000
6	24	0.240000	27.5	0.275000	24	0.240000
7	28.3333	0.283333	28.3333	0.283333	28	0.280000
8	90	0.450000	90	0.450000	90	0.450000
9	49	0.163333	52.5	0.175000	78	0.260000
10	15	0.050000	15	0.050000	15	0.050000

transient faults, we need to align those clones allocated to the PE with the existing clones of the same tasks. Similarly, when the processing time of a task is changed in a clone reconfiguration, the execution states of the clones of the task should be aligned. We call the clones that need alignments the *lost–soul* clones.

Clone alignment is not necessary for pure functional tasks. A pure functional task has no persistent state and its output is determined solely by its input and not by its previous execution states. The clones of a pure functional task are always aligned and ready for execution. In contrast, clone alignment is necessary for the other tasks which are not pure functional. As soon as the state of a lost–soul clone is aligned, the clone must be run periodically to keep pace with the existing clones of the same task. Otherwise, its execution state falls behind and it produces timing errors.

When the alignment of a lost–soul clone cannot be completed in a period, its execution state is behind. One solution for this problem is to stop temporarily the executions of the existing

clones of the same task, copy the state, and then start the executions of the existing clones and the lost–soul clone. If the executions of the existing clones are halted too long, the services provided by these clones may be disrupted for more than one period of time. Because this solution does not guarantee continuous services, it may not be acceptable for hard real–time systems.

5.7.4.1. Two Approaches

There are two basic approaches to clone alignment: the hardware approach and the software approach. Although the hardware approach is transparent to application tasks, it assumes that the replication scheme used is *NMR* (n–modular–redundancy) [4]. All PE's in NMR have an identical set of clones and execute the same instruction simultaneously. The value stored in a memory location is identical for all memories in NMR. In this approach, extra hardwares are built in NMR. When a PE is restarted, the hardwares try to copy the "dirty" memory segments of a non–faulty PE to the memory segments of the restarted PE.

In the software approach, a lost–soul clone is aligned by a *logical* execution state (or a check point). There are two ways to obtain the execution state. The first way is to ask one of the existing clones whose execution states are already aligned to send its execution state and the input leading to this execution state. The second way is to infer the execution state from the inputs and outputs of an existing clone whose state is already aligned. For example, the outputs of some tasks may contain the execution states. Although the second way is not always possible, it is preferred because the existing clones are not disturbed.

5.7.4.2. Align Algorithm

Our align algorithm (Figure 5.8) is based on the software approach. NMR assumed in the hardware approach contradicts with our replication scheme in which PE's may have different sets of clones. The clones allocated to a faulty PE can be re–allocated to non–faulty PE's. The memory contents of PE's may be different.

The input of the align algorithm is a lost–soul clone denoted by C. Because the execution of C may fall behind causing other clones to generate more failure reports, the algorithm suppresses the outputs of C until C is aligned. The outputs are not broadcast to the clones with which C has dependence relations. The align algorithm queues the inputs of C. As discussed earlier, there are two ways to get the execution state of C and the input leading to this state. Regardless how they are obtained, the algorithm identifies this input in the input queue and truncates all elements in the queue preceding and including this input. It installs C with the state and rolls C forward with the truncated queue whose first element is the one following the identified input. The align algorithm tries to give C more than the assigned processing time in a period, if C needs to catch up with the existing clones. Specifically, C may run more than once and consume several inputs in a period. C is executed both at its assigned priority and as a background task. Since background tasks have lowest priorities, they are executed last. When C needs no more background task to consume its accumulated inputs, it is aligned. The state of C and the state of the existing clones have converged. Finally, the algorithm permits the outputs of C to be broadcast.

```
Algorithm align
input: C
begin
        suppress the outputs of C;
        queue the inputs of C;
        get an execution state of C and the corresponding input;
        identify the input in the input queue;
        install C with the execution state;
        roll C forward until it is aligned;
        unsuppress the outputs of C;
end;
```

Figure 5.8: The align algorithm

The rate of convergence depends on the amount of processing time allocated to the background task. If more processing time is allocated, the convergence rate is faster. For example, the convergence rate is doubled, if the background task is assigned three times the processing time of C instead of one.

Example

Table 5.1 shows that a non–pure functional task T^5 has three clones, when the system has four non–faulty PE's. Table 5.5 shows that T^5 has one missing clone to be added to PE_2 and two existing clones whose processing times need to be reduced on PE_4 and PE_6. PE_7 has failed stop. The clone–reconfiguration algorithm first adds the missing clone to PE_2. The align algorithm gets the execution state from one of the existing clones, installs the newly added clone with the state, and rolls the clone forward until there is no accumulated input in the input queue and the state of the clone is aligned. Then, the clone–reconfiguration algorithm reduces the processing time of the clone allocated PE_4. The align algorithm gets the execution state from the clone on PE_2, installs the clone on PE_4 with the state, and rolls the clone forward until its state has converged. Similarly, the processing time of the clone on PE_6 is reduced and the clone is aligned with the clone on either PE_4 or PE_6.

5.7.5. Redundancy and Consistency

The repair server has standby redundancies similar to that of the coordinator. The difference is that while the diagnosis services are atomic, the repair services are not. When the host PE of the active repair server fails, the second element in the chain becomes the new active server as soon as it detects the failure. It notifies the coordinator about the PE failure. If an ongoing repair is terminated prematurely, the new active server starts a new repair with the current clone allocation and $new\text{–}F_r$. In a complete repair service, every non–faulty standby repair server receives $new\text{–}F_r$ from the coordinator in the beginning of the service and F_r from the active repair server in the end. If $new\text{–}F_r$ is not equal to F_r, every non–faulty standby

server knows that there is an incomplete repair.

When the host PE of a standby repair server malfunctions, the next non–faulty standby server in the chain detects and notifies the coordinator about the failure. It assumes the role of the failed standby server. In both cases, a new standby server may be added to the tail of the chain during the repair, if the PE failure is permanent. If the PE failure is transient, the restarted server becomes a standby server and is added to the tail of the chain.

The repair state of the system is characterized by the current clone allocation ($\{x_i^j\}$), the old faulty PE set (F_r), and the new faulty PE set ($new-F_r$). The state is consistent if (1) $new-F_r$ is the sum of F_c's in all clusters, and (2) F_r is an improper subset of $new-F_r$ or vice versa (when faulty PE's are replaced with non–faulty PE's during maintenance). The state is *stable* if F_r is equal to $new-F_r$. All repairs are completed. By broadcasts, $\{x_i^j\}$, F_r, and $new-F_r$ are uniquely defined on all non–faulty repair servers.

5.7.6. Imprecise Clone Reconfiguration

In this section, we show that the clone–reconfiguration algorithm is an imprecise computation. In other words, any partial or incomplete clone re–allocation is an imprecise result of the complete clone re–allocation. The clone allocation produced by an incomplete reconfiguration makes the real–time system more or equally reliable than the original clone allocation before the reconfiguration.

When a clone reconfiguration is interrupted during the first phase of the clone–reconfiguration algorithm, the modify algorithm has not produced the new clone allocation and no clone has been reconfigured. Another clone reconfiguration can be started as though no clone reconfiguration has been performed. Hence, the reliability of the system remains the same during the first phase.

When a clone reconfiguration is interrupted during the second phase of the clone–reconfiguration algorithm, some clones may be partially reconfigured. Specifically, the clone–

reconfiguration algorithm sends the new clone allocation $\{new\text{-}x_i^j\}$ iteratively to each non-faulty PE_i. The iterative sending process may be aborted. We show that each iteration improves the reliability of the current clone allocation. Let's define some terms first.

When the active server receives a positive acknowledgement from PE_i during the i^{th} iteration, it knows that PE_i has installed the new clone allocation. We can ignore the case in which the processing time of a clone is changed from a non-zero value to another non-zero value. Since the number of the existing clones is not changed, the reliability of the system is not affected. Hence, we only concentrate on the other two cases in which the processing time of a clone is changed from zero to a non-zero value (addition) or from a non-zero value to zero (deletion). Let e be the number of the existing clones of a task T^j before the clone reconfiguration. c is the number of the clones required for T^j. e' is the current number of the existing clones of T^j at the completion of the i^{th} iteration. We know that the modify algorithm does not migrate the existing clones from a non-faulty PE to another. We show that $|e'-c|$ is less than or equal to $|e-c|$.

When $e=c$, T^j does not have either excessive clones nor missing clones. Since the modify algorithm does not migrate the existing clones, the reliability of T^j remains the same during the clone reconfiguration.

When $e>c$, T^j has excessive clones to be removed from the existing clone allocation. Each iteration deletes at most an excessive clone of T^j from a PE. It decreases the number of the T^j clones by one or zero. Although the number of the T^j clones decreases monotonically during the second phase, e' is always greater than or equal to c. The current clone allocation always meets the reliability requirement of T^j.

When $e<c$, T^j has missing clones to be added to the existing clone allocation. Each iteration adds at most a missing clone of T^j to a PE. It increases the number of the T^j clones at most by one. Since each iteration increases the number of the T^j clones monotonically, e' is always greater than or equal to e. Hence, the reliability of T^j is improved gradually. The

current clone allocation is no less reliable than the original clone allocation before the clone reconfiguration.

5.8. Summary

We have presented the algorithms for error detection, failure diagnosis, and clone reconfiguration in replicated hard real–time systems. Our approach is to use active replication for error detection and masking, and imprecise computation to determine the new configuration after one or several PE failures. Our approach has several advantages. First, imprecise repairs improve the reliability of the current clone allocation monotonically and can be aborted any time when there is a new PE failure. Failure reporting and repairs are event– and demand–driven, and thus are more responsive and efficient. Second, our approach uses the high–level knowledge such as the result comparisons and error indicators in the application level and timing information for error detections. It is capable of detecting a larger class of faults. So, only those PE's providing the diagnosis and repair services need to be fail–stop, fail–fast. Third, since each clone reports its detected errors independently, the detection of errors is more reliable and is capable of detecting errors caused by transient faults.

Chapter 6
Conclusion

Hard real–time systems have timing requirements which must be satisfied to ensure their correctness and integrity. The computations in many of these systems are modeled as a dependence graph of parallel real–time tasks. The fulfillment of timing requirements in these systems hinges on the proper scheduling of these tasks by the system scheduler.

In this thesis, we discuss the problem of scheduling parallel real–time tasks on a multiprocessor system so that the system has predictable performance and will produce acceptable results in time. Three techniques are employed to alleviate this difficult scheduling problem: the imprecise computation technique, parallel processing, and the redundancy and masking technique. These techniques prevent timing faults, achieve graceful degradation, and make real–time systems more flexible and hence predictable. The imprecise computation technique allows a real–time task to produce an imprecise result. When insufficient computational resources are available, a task may not have enough resources to complete the execution before the deadline. In such cases, the system scheduler tries to complete at least the hard subtask of the task to produce an acceptable result. When the sequential execution time of a real–time task is longer than its required response time, parallel processing may be used to shorten its execution time to prevent timing faults. Real–time system designers can utilize a number of parallel algorithms designed for a variety of powerful and cost–effective multiprocessors. A predictable real–time system must tolerate certain hardware or software faults. Because real–time systems have stringent timing constraints, we employ the redundancy and masking technique for fault tolerance.

Based on these three techniques, we extend the traditional task model to the parallelizable imprecise computation model and the replicated imprecise computation model. We design

several efficient multiprocessor scheduling algorithms for these imprecise computation models. The results are summarized in the next section.

6.1. Summary of Results

In Chapter 3, we study the problem of scheduling parallelizable imprecise tasks on a multiprocessor system. Recent advances in parallel processing technology improve the execution speed of a task by decomposing it into a number of subtasks which are executed concurrently on distinct processors. The capability of shortening task execution time increases the likelihood that real–time applications meet their timing constraints. For systems with occasional overloads, it may not be always possible to meet the timing constraints. One possible approach to avoiding such timing faults is to use the imprecise computation model. In our model, each imprecise task may be parallelized and executed on multiple processors with a multiprocessing overhead which is assumed to be a linear function of the degree of parallelism. We show that the problem of time allocation in such a real–time application can be formulated and solved as a linear programming problem. We also present an algorithm for constructing a multiprocessor schedule from the linear programming solution. This algorithm reduces the multiprocessing overhead generated in the multiprocessor schedule, while guaranteeing that the multiprocessing overhead does not exceed a linear upper bound. The worst–case time complexity of the problem is $O(n^6)$ where n is the number of tasks.

In Chapter 4, we study the problem of scheduling periodic tasks on a multiprocessor system under a fault–tolerant requirement. Our approach incorporates both the redundancy and masking technique and the imprecise computation model. Since the tasks in hard real–time systems have stringent timing constraints, the redundancy and masking technique is more appropriate than the rollback and retry technique which usually requires extra time for error recovery. The imprecise computation model provides flexible functionality by trading off the quality of the result produced by a task with the amount of processing time required to produce it. In case of

hardware failures, it permits the performance of a real–time system to degrade gracefully. The objective of the scheduling problem is to find for each periodic task (a) the amount of its processing time and (b) the assignment of its replicated clones to the PE's so that the total weighted utilization (processing time / period) or *TWU* of all periodic tasks is maximized. We design two algorithms for solving the problem. The allocate algorithm seeks the near–optimal task allocations. The modify algorithm seeks the task allocations which maintain execution continuity with respect to the previous task allocation, while trying to maximize *TWU*. Maintaining execution continuity reduces the transient errors due to task migrations and the overloading of processors and memories. It also permits incremental recovery. Our analytical and simulation results show that both the allocate algorithm and the modify algorithm are resilient under hardware failures. The allocate algorithm oftentimes generates optimal solutions, while it seldom produces infeasible task allocations. In almost all cases, the modify algorithm produces a *TWU* slightly less than the *TWU* produced by the allocate algorithm.

6.2. Directions for Future Research

In this thesis, the objective function of the scheduling problems is to minimize the total weighted error ($\min \sum_{j=1}^{n} w^j \epsilon^j$). For some real–time applications, it is better to minimize the maximum weighted error ($\min \max_{j=1}^{n} w^j \epsilon^j$). This can be accommodated in a straight forward manner for the imprecise multiprocessor scheduling problem discussed in Chapter 3. We need only to change the objective function of the LP and add one more inequality constraint to it, as shown below.

$$\min \epsilon_{\max}$$
$$\epsilon_{\max} \geq w^j \epsilon^j \qquad j = 1, 2, \ldots, n$$

The time complexity of the new LP is $O(n^6)$, the same as the old LP. For the replicated imprecise task allocation problem, the change is not so straight forward. A new greedy algorithm that minimizes the maximum error is needed.

Our preliminary study shows that the multiprocessing overheads due to memory interference, cache coherence, and network contention and blocking are nonlinear functions of the degree of parallelism. Furthermore, they may not be represented and bounded closely by linear functions. We plan to extend the linear programming approach in three different directions to incorporate nonlinear multiprocessing overheads.

When multiprocessing overheads are convex functions of the degree of parallelism, convex programming techniques can be similarly applied to the convex programming formulation of the imprecise multiprocessor scheduling problem. Although the convexity properties guarantee the convergence of search algorithms and the local optimal point identical to the global optimal point, the rate of the convergence of search algorithms may be slow. Generally, the rate of convergence depends on the search techniques used and the convexity of the objective function and the feasible regions of a problem instance.

Branch and bound algorithms are used in the second direction to solve the imprecise multiprocessor scheduling problem in which the multiprocessing overheads are constrained by nonlinear and integer functions. Our linear programming approach may provide a good initial incumbent and a lower bound for branch and bound algorithms. This is important because the neighborhoods of the imprecise multiprocessor scheduling problem with nonlinear multiprocessing overheads may be too small for branch and bound algorithms to search for a globally good solution from totally random initial solutions. The drawback is that branch and bound algorithms are enumerative in nature and may not converge to an optimal solution. The third direction is a mixed strategy approach composed of convex programming techniques and branch and bound techniques.

One approach to coping with software bugs is n–version programming [7]. In n–version programming, the versions of a task have diverse designs and implementations, and hence they may be assigned with different weights. Similar to the replicated imprecise task allocation problem in Section 4.3, we formulate the n–version imprecise task allocation problem in which the objective is defined as $\sum_{C_i^j \in NPTS} w_i^j u_i^j$. *NPTS* denotes a *PTS* in which each periodic task may have several different versions that may have distinct hard tasks and distinct soft tasks.

To avoid allocating the versions of a task to the same PE, the versions are grouped as a unit and scheduled consecutively. Since the versions of a task are different, we may characterize them by the average utilization and the variance. The variance can be either $\sum_{i=1}^{m} |u_i^j - \bar{u}|$ or $\sum_{i=1}^{m} (u_i^j - \bar{u})^2$. Without loss of generality, we may assume each task has m versions. $m - c^j$ of these versions have $u_i^j = 0$. If a task has a large average utilization and greater than 1 but less than m versions, the variance of the task is probably large.

Similar to the LUF algorithm in Section 4.4, the *largest variance first* (*LVF*) algorithm can be designed. The difference between these two algorithms is the order in which the tasks are arranged. The LVF algorithm sorts the tasks in the decreasing order of the variance of utilization. When the tasks have the same variance, they are arranged in the decreasing order of average utilization. When the tasks have the same average and variance, they are arranged in the non–increasing number of versions. The tasks are so ordered that the processor utilizations of the task allocation generated by the LVF algorithm have a small variance. The n–version imprecise task allocation problem is more general than the replicated imprecise task allocation. Hence, it is more difficult to prove the optimality of the LVF algorithm. The performance of this algorithm can be evaluated by computer simulations.

Our approach to maintaining the execution continuity of n–version periodic tasks is similar to that for replicated periodic tasks. In Section 4.5, we modify the LUF algorithm to get the

FCLUF algorithm. We make the similar modification to the LVF algorithm to obtain the FCLVF algorithm. Then, we change the FCLUF algorithm called in the modify algorithm in Section 4.5 to the FCLVF algorithm.

Appendix A

Proof of Theorem 4.1: We first show that any replicated periodic task system *RPTS* that violates the condition $R(LUF) \leq 4/3 - 1/3m + (c-1)d/mZ^n_{(m)}(OPT)$ must have $u^n > Z^n_{(m)}(OPT)/3$. Suppose a *RPTS*, if scheduled by the LUF algorithm, violates the above condition. Figure A.1 depicts the allocation of the clones in *RPTS* to the PE's. If T^j is the first task to violate the condition, then we can delete $T^{j+1}, T^{j+2}, \ldots, T^n$ from *RPTS*. The new *RPTS* with less tasks also violates the condition. Thus, we can assume without the loss of generality that it is always the last task which violates the condition.

Let $C^n_{c^n}$ be the last clone in *RPTS*. $s = Z^n_{(m)}(LUF) - u^n$. Figure A.2 shows that s may be larger than the processor utilizations of some PE's that have the clones of the same task. d denotes the maximum of such differences between s and the processor utilizations of these PE's.

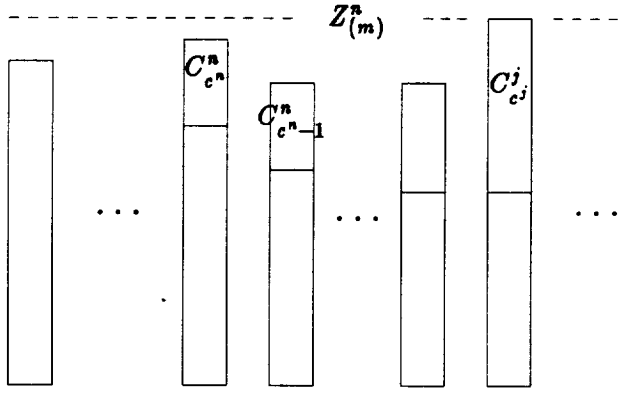


Figure A.1: An LUF task allocation

If s is less than the processor utilizations of these PE's, d is 0.

$$Z^n_{(m)}(OPT) \geq \frac{1}{m}\sum_{j=1}^{n} c^j u^j$$

$$\sum_{j=1}^{n} c^j u^j - u^n \geq ms - (c-1)d$$

The first inequality asserts that the optimal maximum processor utilization must not be less than the lower bound, the average processor utilization. The second inequality states that when all but the last clone ($C^n_{c^n}$) are scheduled, every PE has a processor utilization no less than s except some PE's that have the clones of the task T^n. The number of these PE's is not more than $c-1$ and their utilizations are not less than $s-d$.

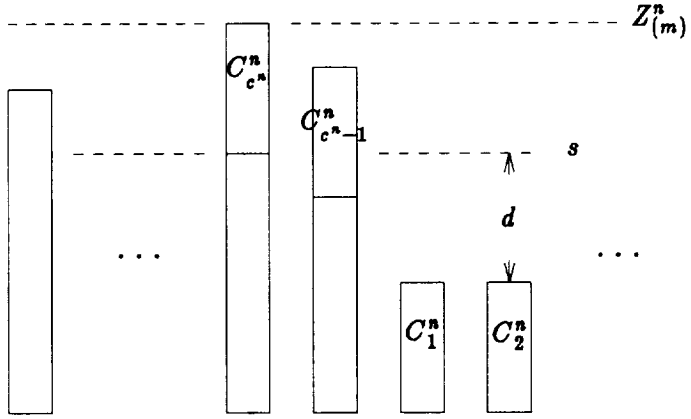


Figure A.2: A task allocation violating the condition

$$\frac{Z^n_{(m)}(LUF)}{Z^n_{(m)}(OPT)} = \frac{s+u^n}{Z^n_{(m)}(OPT)} \leq \frac{(m-1)u^n}{mZ^n_{(m)}(OPT)} + \frac{\sum_{j=1}^{n} c^j u^j}{mZ^n_{(m)}(OPT)} + \frac{(c-1)d}{mZ^n_{(m)}(OPT)}$$

$$1 + \frac{(m-1)u^n}{mZ^n_{(m)}(OPT)} + \frac{(c-1)d}{mZ^n_{(m)}(OPT)} > \frac{4}{3} - \frac{1}{3m} + \frac{(c-1)d}{mZ^n_{(m)}(OPT)}$$

$$u^n > \frac{Z^n_{(m)}(OPT)}{3}$$

After some manipulation of the inequalities, we show $u^n > Z^n_{(m)}(OPT)/3$. That is, at most two clones are allocated to a PE in the optimal task allocation $\mathbf{A}(OPT)$, if the condition is violated. We use two operations to rearrange the clones in $\mathbf{A}(OPT)$ so that the new $\mathbf{A}'$ has a special pattern. Of the two clones assigned to a PE, the first operation moves the clone with a larger utilization to the bottom level. The second operation exchanges the two top–level tasks of any two PE's so that the maximum processor utilization is reduced. For example, Figure A.3 shows $u_i \geq u_j$ and $u_k \geq u_l$. If C_i and C_l are the clones of two different tasks and so are C_k and C_j, then C_k and C_l can be exchanged. The maximum processor utilization of the two PE's is reduced. Because the first operation does not change the processor utilization of a PE and the second operation always reduces the maximum processor utilization of any two PE's, they do not increase the optimal maximum processor utilization of $RPTS$.

We apply these two operations to $\mathbf{A}(OPT)$ for a finite number of times until further applications do not reduce the maximum processor utilization. We reindex the PE's or exchange the columns of $\mathbf{A}'$ so that the clones in the bottom level are in non–decreasing order (i.e., $u_1 \geq \cdots \geq u_m$). The clones in the top layer are in non–increasing order (i.e., $u_{m+1} \geq \cdots \geq u_{2m}$). Figure A.3 depicts the only exception in which $u_i \geq u_j$, $u_k > u_l$, and C_k and C_j are the clones of the same task. Furthermore, one of these clones (i.e., C_k, C_j, and their sibling clones) is allocated to the mth PE. It is not possible to have $u_i \geq u_j$, $u_k > u_l$, and C_i and C_j are the clones of the same task. Since reindexing the PE's does not change the maximum

processor utilization, the reindexed $\mathbf{A}''$ is optimal.

We claim that the $\mathbf{A}''$ is isomorphic to the task allocation produced by LUF, $\mathbf{A}(LUF)$. There are two cases. Graham has proved the case when there is no exception as displayed in Figure A.3 [26]. When there is an exception, $\mathbf{A}''$ is also isomorphic to $\mathbf{A}(LUF)$. There exist one–to–one mappings for the clones of the same utilizations. Hence, $\mathbf{A}(LUF)$ is optimal, $R(LUF)=1$, and we have shown the contradiction. Therefore, $R(LUF)\leq 4/3-1/3m+(c-1)d/mZ^{\pi}_{(m)}(OPT)$ for any $RPTS$.

The condition $R(LUF)\leq 4/3-1/3m+(c-1)d/mZ^{\pi}_{(m)}(OPT)$ can be simplified. We have shown that if $R(LUF)>4/3-1/3m+(c-1)d/mZ^{\pi}_{(m)}(OPT)$, then $u^{\pi}>Z^{\pi}_{(m)}(OPT)/3$. $u^{\pi}+s=Z^{\pi}_{(m)}(OPT)$. $s<2Z^{\pi}_{(m)}(OPT)/3$. $d<Z^{\pi}_{(m)}(OPT)/3$ because those PE's assigned with the clones of T^{π} have utilizations no less than u^{π}. Hence, $R(LUF)>4/3-1/3m+(c-1)d/mZ^{\pi}_{(m)}(OPT)$ implies $R(LUF)>4/3+(c-2)/3m$. Take the contrapositive, $R(LUF)\leq 4/3+(c-2)/3m$ implies

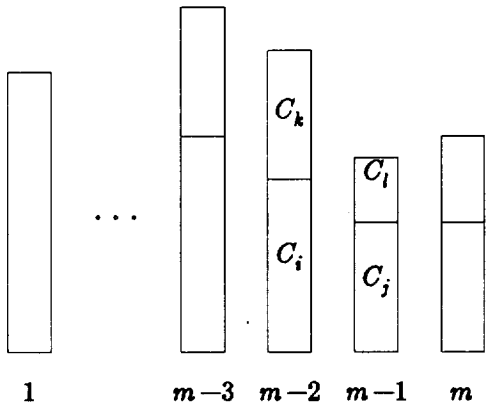


Figure A.3: Task allocation generated by LUF

$R(LUF) \leq 4/3 - 1/3m + (c-1)d/mZ^n_{(m)}(OPT)$. The condition is simplified to $R(LUF) \leq 4/3 + (c-2)/3m$. □

Proof of Theorem 4.2: The theorem is proved in two parts. In the first part, we show that the sequence $D^0(LUF) \cdots D^n(LUF)$ converges to a small finite number. We shall omit LUF from the notations whenever the meaning is clear and unambiguous from the context. In the second part, we show that D^n converges to 0 almost surely.

$$Z^n_{(m)} = \max(Z^{n-1}_{(m)}, Z^{n-1}_{(c^n)} + u^n)$$

$$D^n = Z^n_{(m)} - \frac{1}{m}\sum_{i=1}^{m} Z^n_{(i)}$$

$$D^n = \max(Z^{n-1}_{(m)} - \frac{1}{m}\sum_{i=1}^{m} Z^n_{(i)},\ Z^{n-1}_{(c^n)} + u^n - \frac{1}{m}\sum_{i=1}^{m} Z^n_{(i)})$$

$$= \max(D^{n-1} - \frac{c^n u^n}{m},\ D^{n-1} - (Z^{n-1}_{(m)} - Z^{n-1}_{(c^n)}) + u^n - \frac{c^n u^n}{m})$$

The difference between the total processor utilizations before and after the allocation of the clones of task T^n is $\sum_{i=1}^{m} Z^n_{(i)} - \sum_{i=1}^{m} Z^{n-1}_{(i)} = c^n u^n$. Hence, $D^n > D^{n-1}$ if and only if $Z^{n-1}_{(c^n)} > Z^{n-1}_{(m)} - (m-c^n)u^n/m$. Moreover, the increment is bounded by $(m-c^n)u^n/m$. Otherwise, D^{n-1} is decremented at most by $c^n u^n/m$.

Suppose D^{n-1} is incremented. $Z^n_{(m)} - Z^{n-1}_{(c^n)} = u^n$. Since $(m-c^n)u^n/m > Z^{n-1}_{(m)} - Z^{n-1}_{(c^n)}$, $Z^n_{(m)} - Z^n_{(k^n)} > c^n u^n/m$ where $m - c^n \leq k^n < m$. $(0 < m - k^n \leq c^n.)$ Figure A.4 depicts the ordering relations among the processor utilizations of some PE's before and after the clones of task T^n are scheduled. D^n is incremented again, if

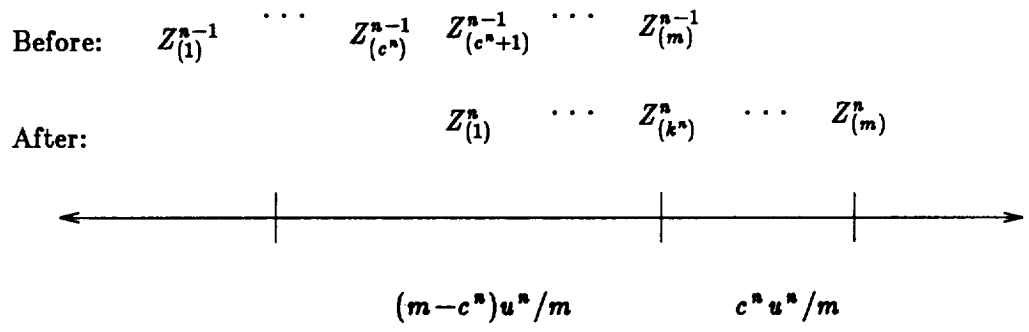


Figure A.4: Order relations of processor utilizations

$$Z^n_{(c^{n+1})} > Z^n_{(m)} - \frac{(m-c^{n+1})u^{n+1}}{m}$$

$$Z^n_{(c^{n+1})} - Z^n_{(k^n)} > Z^n_{(m)} - Z^n_{(k^n)} - \frac{(m-c^{n+1})u^{n+1}}{m}$$

$$> \frac{c^n u^n}{m} - \frac{(m-c^{n+1})u^{n+1}}{m}$$

The last inequality shows that it is unlikely for two increments to occur consecutively. Since $u^n \geq u^{n+1}$, the right hand side of the last inequality is positive when $c^{n+1} \geq m - c^n$. When $k^n \geq c^{n+1}$, the left hand side is negative. Because $k^n \geq m - c^n$, the left hand side is negative when $c^{n+1} \leq m - c^n$.

When n is large, the condition for increments becomes harder to satisfy as $(m-c^n)u^n/m$ approaches to zero. The probability of decrements gets higher. Moreover, the increment is bounded by a small number $(m-c^n)u^n/m$. If the condition for increments is satisfied, there are more than $m - c^n$ PE's whose processor utilizations lie in the small interval between $Z^{n-1}_{(m)}$ and $Z^{n-1}_{(m)} - (m-c^n)u^n/m$. Hence, D^n converges to a small finite number when n is large.

In the second part, we expand D^n and obtain the following expression,

$$D^n = \max(Z_{(m)}^{n-1} - \frac{1}{m}\sum_{i=1}^{m} Z_{(i)}^{n},\ Z_{(c^n)}^{n-1} - \frac{1}{m}\sum_{i=1}^{m} Z_{(i)}^{n} + u^n)$$

$$= \max(D^{n-1} - \frac{c^n u^n}{m},\ Z_{(c^n)}^{n-1} - \frac{1}{m}\sum_{i=1}^{m} Z_{(i)}^{n-1} + u^n - \frac{c^n u^n}{m})$$

$$= \max(D^{n-1},\ Z_{(c^n)}^{n-1} - \frac{1}{m}\sum_{i=1}^{m} Z_{(i)}^{n-1} + u^n) - \frac{c^n u^n}{m}$$

$$= \cdots$$

$$= \max_{1 \le j \le n} \left\{ Z_{(c^j)}^{j-1} - \frac{1}{m}\sum_{i=1}^{m} Z_{(i)}^{j-1} + u^j - \sum_{i=j}^{n} \frac{c^i u^i}{m} \right\}.$$

We have shown in the first part that the sequence $D^0 \cdots D^n$ converges to a small finite number, when n is large. $D^{j-1} = Z_{(m)}^{j-1} - \frac{1}{m}\sum_{i=1}^{m} Z_{(i)}^{j-1} \ge Z_{(c^j)}^{j-1} - \frac{1}{m}\sum_{i=1}^{m} Z_{(i)}^{j-1}$ in the maximization expression. Under the assumption stated in the theorem, it has been shown [23] for an appropriate choice of $n' < n$ that if $1 \le j \le n'$,

$$\lim_{n\to\infty} u^j - \sum_{i=j}^{n} \frac{u^i}{m} = -\infty. \text{ (a.s.)}$$

If $n' \le j \le n$, for every $\delta > 0$, ϵ in $(0, \epsilon)$ can be chosen in such a way that

$$\lim_{n\to\infty} u^j - \sum_{i=j}^{n} \frac{u^i}{m} \le \delta. \text{ (a.s.)}$$

Therefore, $\lim_{n\to\infty} D^n(LUF) = 0$ (a.s.). □

References

[1] "Using the Sequent Balance 8000," ANL/MCS–TM–66, Mathematics and Computer Science, Argonne National Laboratory, 1986.

[2] "Using the Encore Multimax," ANL/MCS–TM–65, Mathematics and Computer Science, Argonne National Laboratory, 1986.

[3] *iPSC System Overview,* Intel Corporation, Nov. 1986.

[4] S. Adams and T. Sims, "A Tagged Memory Technique for Recovery from Transient Error in Fault Tolerant Systems," *Proc. 11th Real–Time Systems Symp.*, pp. 312–321, IEEE, Dec. 1990.

[5] S. G. Akl, *The Design and Analysis of Parallel Algorithms,* Prentice–Hall, Englewood Cliffs, NJ, 1989.

[6] T. Anderson and J. C. Knight, "A Framework for Software Fault Tolerance in Real–Time Systems," *Trans. on Software Eng.*, vol. SE–9, no. 3, pp. 355–364, IEEE, May 1983.

[7] A. Avizienis and J. C. Laprie, "Dependable Computing: From Concept to Design Diversity," *Proc. of the IEEE*, vol. 74, no. 5, pp. 629–638, May 1986.

[8] O. Babaoglu and R. Drummond, "Streets of Byzantium: Network Architectures for Fast Reliable Broadcasts," *Trans. on Software Eng.*, vol. SE–11, no. 6, pp. 546–554, IEEE, June 1985.

[9] J. Bannister and K. Trivedi, "Task Allocation in Fault–Tolerant Distributed Systems," *Acta Informatica*, vol. 20, pp. 261–281, 1983.

[10] J. Blazewicz, M. Drabowski, and J. Weglarz, "Scheduling Multiprocessor Tasks to Minimize Scheduling Length," *Trans. on Computers*, vol. C–35, no. 5, pp. 389–393, May 1986.

[11] J. Blazewicz, "Selected Topics in Scheduling Theory," *Annals of Discrete Math.*, vol. 31, pp. 1–60, 1987.

[12] P. Buneman, S. Davidson, and A. Watters, "A Semantics for Complex Objects and Approximate Queries," *Proc. 7th Symp. Principles of Database Systems*, pp. 305–314, 1988.

[13] J. M. Chang and N. F. Maxemchuk, "Reliable Broadcast Protocols," *ACM Trans. on Computer Systems*, vol. 2, no. 3, pp. 251–273, August 1984.

[14] J. Y. Chung, J. W. Liu, and K.–J. Lin, "Scheduling Periodic Jobs That Allow Imprecise Results," *Trans. on Computers*, vol. C–39, no. 9, pp. 1156–1174, IEEE, Sep. 1990.

[15] E. G. Coffman, Jr. (ed.), *Computer and Job–Shop Scheduling Theory*, Wiley, New York, NY, 1976.

[16] E. G. Coffman, M. R. Garey, and D. S. Johnson, "Approximation Algorithms for Bin–Packing – An Updated Survey," in *Algorithm Design for Computer System Design*, ed. G. Ausiello, M. Lucertini, and P. Serafini, pp. 49–106, Springer–Verlag, 1984.

[17] F. Cristian, "Understanding Fault–Tolerant Distributed Systems," *Comm. of the ACM*, vol. 34, no. 2, pp. 56–78, 1991.

[18] S. K. Dhall and C. L. Liu, "On a Real–Time Scheduling Problem," *Oper. Res.*, vol. 26, no. 1, pp. 127–140, Jan. 1978.

[19] D. Dolev and M. Warmuth, "Profile Scheduling of Opposing Forests and Level Orders," *SIAM J. of Algorithm and Discrete Mathematics*, vol. 6, no. 4, pp. 665–687, Oct. 1985.

[20] J. Du and J. Leung, "Complexity of Scheduling Parallel Task Systems," Technical Report UTDCS 6–87, Univ. Texas at Dallas, Dallas, TX, 1987.

[21] P. Ezhilchelvan and R. de Lemos, "A Robust Group Membership Algorithm for Distributed Real–Time Systems," *Proc. 11th Real–Time Systems Symp.*, pp. 173–179, IEEE, Dec. 1990.

[22] S. French, *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, John Wiley and Sons, 1982.

[23] J. B. Frenk and A. H. G. Rinnooy Kan, "The Asymptotic Optimality of the LPT Rule," *Math. Oper. Res.*, vol. 12, no. 2, pp. 241–254, May 1987.

[24] D. Gannon and W. Jalby, "The Influence of Memory Hierarchy on Algorithm Organization: Programming FFTs on a Vector Multiprocessor," in *The Characteristics of Parallel Algorithms*, ed. L. Jamieson, D. Gannon, and R. Douglass, pp. 277–301, MIT Press, 1987.

[25] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Freeman, San Francisco, CA, 1979.

[26] R. L. Graham, "Bounds on the Performance of Scheduling Algorithms," in *Computer and Job-Shop Scheduling Theory*, ed. E. G. Coffman, Jr., pp. 165–227, John Wiley & Sons, 1976.

[27] C. C. Han and K.–J. Lin, "Scheduling Parallelizable Real–Time Jobs on Multiprocessors," *Proc. 10th Real-Time Systems Symp.*, pp. 59–67, IEEE, Dec. 1989.

[28] W. Harrod, "Parallel Programming with the BLAS," in *The Characteristics of Parallel Algorithms*, ed. L. Jamieson, D. Gannon, and R. Douglass, pp. 253–275, MIT Press, 1987.

[29] P. Harter, Jr., "Response Times in Level–Structured Systems," *Trans. Computer Systems*, vol. 5, no. 3, pp. 232–248, ACM, Aug. 1987.

[30] S. Horiguchi and T. Nakada, "Experimental Performance Evaluation of Parallel Fast Fourier Transform on a Multiprocessor Workstation," *Proc. Int'l Conf. on Parallel Processing*, vol. 3, pp. 97–101, IEEE, 1990.

[31] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms,* Computer Science Press, Rockville, MD, 1978.

[32] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw–Hill, 1984.

[33] O. Ibarra and C. Kim, "Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems," *JACM*, vol. 22, no. 4, pp. 463–468, 1975.

[34] A. K. Jones and E. F. Gehringer (ed.), "Cm* Multiprocessor Project : A Research Review," Tech. Rept. CMU–CS–80–131, Carnegie–Mellon University, Pittsburgh, PA, July 1980.

[35] R. Jurgen, "Smart Cars and Highways Go Global," *Spectrum*, pp. 26–36, IEEE, May 1991.

[36] N. Karmarkar, "A New Polynomial–time Algorithm for Linear Programming," *Combinatorica*, vol. 4, pp. 373–395, 1984.

[37] L. G. Khachian, "Polynomial Algorithms in Linear Programming," *Zhurnal Vychislitelnoi Mathematiki i Matematicheskoi Fiziki*, vol. 20, pp. 53–72, 1980.

[38] K. Kim and B. Min, "Approaches to Implementation of Multiple DRB Stations in Tightly–Coupled Computer Networks," *Proc. 15th Int'l Computer Software and Applications Conf.*, pp. 550–557, Sep. 1991.

[39] K. H. Kim and H. O. Welch, "Distributed Execution of Recovery Block: An Approach for Uniform Treatment of Hardware and Software Faults in Real–Time Applications," *Trans. on Computers*, vol. C–38, no. 5, pp. 626–636, IEEE, May. 1989.

[40] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwalbl, C. Senft, and R. Zainlinger, "Distributed Fault–Tolerant Real–Time Systems: the MARS Approach," *Micro*, pp. 25–40, IEEE, Feb. 1989.

[41] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger, "Tolerating Transient Faults in MARS," *Proc. 20th Int'l Symp. on Fault–Tolerant Computing*, pp. 466–473, IEEE, Jun. 1990.

[42] C. M. Krishna and K. G. Shin, "On Scheduling Tasks with a Quick Recovery from Failure," *Trans. on Computers*, vol. C–35, no. 5, pp. 448–454, IEEE, May 1986.

[43] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh, "Parallel Supercomputing Today and the Cedar Approach," *Science*, vol. 231, pp. 967–974, Feb. 1986.

[44] E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, "Recent Developments in Deterministic Sequencing and Scheduling: A Survey," in *Deterministic and Stochastic Scheduling*, ed. J. K. Lenstra, M. A. H. Dempster, and A. H. G. Rinnooy Kan, pp. 285–298, 1974.

[45] Eugene L. Lawler, "Fast Approximation Algorithms For Knapsack Problems," *Math. of Operations Research*, vol. 4, no. 4, pp. 339–356, Nov. 1979.

[46] E. L. Lawler and C. U. Martel, "Scheduling Periodically Occurring Tasks on Multiple Processors," *Information Processing Letters*, vol. 12, no. 1, pp. 9–12, Feb. 1981.

[47] Y. Lee and C. Krishna, "Optimal Scheduling of Signature Analysis for VLSI Testing," *Trans. on Computers*, vol. C–40, no. 3, pp. 336–341, IEEE, Mar. 1991.

[48] J. Lehoczky, "Fixed Priority Scheduling of Periodic Task Set with Arbitrary Deadlines," *Proc. 11th Real-Time Systems Symp.*, pp. 201–209, IEEE, Dec. 1990.

[49] J. Lehoczky, L. Sha, J. Strosnider, and H. Tokuda, "Fixed Priority Scheduling Theory for Hard Real-Time Systems," in *Foundations of Real-Time Computing: Scheduling and Resource Management*, ed. Andre van Tilborg and Gary Koob, pp. 1–30, Kluwer Academic Publishers, 1991.

[50] J. K. Lenstra and A. H. G. Rinnooy Kan, "Scheduling Theory since 1981: an Annotated Bibliography," Report No. 188/83, Mathematisch Centrum, Amsterdam, the Netherlands, 1983.

[51] J. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks," *Performance Evaluation*, vol. 2, pp. 237–250, 1982.

[52] J. Leung, T. Tam, C. Wong, and G. Young, "Minimizing Mean Flow Time with Error Constraint," *Proc. 10th Real-Time Systems Symp.*, pp. 1–11, IEEE, Dec. 1989.

[53] J. Leung and C. Wong, "Minimizing the Number of Late Tasks with Error Constraint," *Proc. 11th Real-Time Systems Symp.*, pp. 32–40, IEEE, Dec. 1990.

[54] K. Lin and J. Gannon, "Atomic Remote Procedure Call," *Trans. on Software Eng.*, vol. SE–11, no. 10, pp. 1126–1135, IEEE, Oct. 1985.

[55] K. Lin, S. Natarajan, J. Liu, and T. Krauskopf, "Concord: A System of Imprecise Computations," *Proc. 11th Computer Software and Application Conference*, pp. 75–81, Jul. 1987.

[56] K.–J. Lin, S. Natarajan, and J. W. Liu, "Imprecise Results: Utilizing Partial Computations in Real–Time Systems," *Proc. 8th Real–Time Systems Symp.*, pp. 210–217, IEEE, Dec. 1987.

[57] C. L. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real–Time Environment," *J. of the ACM*, vol. 10, no. 1, pp. 46–61, Jan. 1973.

[58] J. Liu, K. Lin, W. Shih, A. Yu, J. Chung, and W. Zhao, "Algorithms for Scheduling Imprecise Computations," *Computer*, pp. 58–68, IEEE, May 1991.

[59] J. Liu, K. Lin, and C. Liu, "Issues in the Imprecise Computation Approach to Fault Tolerance," *Systems Design Synthesis Technology Workshop*, pp. 269–286, Sep. 1991.

[60] E. L. Lloyd, "Critical Path Scheduling with Resource and Processor Constrains," *J. of ACM*, vol. 29, no. 3, pp. 781–811, 1982.

[61] R. McNaughton, "Scheduling with Deadlines and Loss Functions," *Management Science*, vol. 12, pp. 1–12, 1959.

[62] A. Mok and M. Dertouzos, "Multiprocessor Scheduling in a Hard Real–Time Environment," *Proc. 7th Texa Conf. Comput. Symp.*, pp. 5–12, Nov. 1978.

[63] S. Natarajan and K.–J. Lin, "FLEX: Toward Flexible Real–Time Programs," *Proc. Int'l Conference on Computer Languages '88*, pp. 272–193, Oct. 1988.

[64] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu, "Medusa: An Experiment in Distributed Operating System Structure," *Comm. of the ACM*, vol. 23, no. 2, pp. 92–104, Feb. 1980.

[65] D. A. Padua and M. J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Comm. of the ACM*, vol. 29, no. 12, pp. 1184–1201, Dec. 1986.

[66] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice–Hall, 1982.

[67] R. Rajkumar, "Real–Time Synchronization Protocols for Shared Memory Multiprocessors," *Proc. 10th Int'l Conf. on Distributed Computing Systems*, pp. 116–123, IEEE, May 1990.

[68] K. Ramamritham, "Allocation and Scheduling of Complex Periodic Tasks," *Proc. 10th Int'l Conf. on Distributed Computing Systems*, pp. 108–115, IEEE, May 1990.

[69] P. Ramanathan, K. Shin, and R. Butler, "Fault–Tolerant Clock Synchronization in Distributed Systems," *Computer*, pp. 33–42, IEEE, Oct. 1990.

[70] R. Schlichting and F. Schneider, "Fail–Stop Processors: an Approach to Designing Fault–Tolerant Computing Systems," *Trans. Computer Systems*, pp. 222–238, ACM, Aug. 1983.

[71] F. Schneider, "Byzantine Generals in Action: Implementing Fail–Stop Processors," *Trans. Computer Systems*, vol. 2, no. 2, pp. 145–154, ACM, May 1984.

[72] F. Schneider, "Implementing Fault–Tolerant Services Using the State Machine Approach: A Tutorial," *Computing Surveys*, vol. 22, no. 4, pp. 299–319, ACM, Dec. 1990.

[73] C. L. Seitz, "The Cosmic Cube system," *Comm. of the ACM*, vol. 28, no. 1, pp. 22–33, Jan. 1985.

[74] L. Sha, J. Lehoczky, and R. Rajkumar, "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *Proc. 7th Real–Time Systems Symp.*, pp. 181–191, IEEE, Dec. 1986.

[75] L. Sha, R. Rajkumar, K. Ramamritham, and J. Lechoczky, "Mode Change Protocols for Priority Driven Real–Time Scheduling," *Real–Time Systems Journal*, vol. 1, pp. 243–264, 1989.

[76] A. Shaw, "Reasoning about Time in Higher Level Language Software," *Trans. on Software Eng.*, vol. 15, no. 7, pp. 875–889, IEEE, Jul. 1988.

[77] W. K. Shih, J. W. Liu, and J. Y. Chung, "Fast Algorithms for Scheduling Imprecise Computations," *Proc. 10th Real-Time Systems Symp.*, pp. 12–19, IEEE, Dec. 1989.

[78] W. K. Shih, J. W. S. Liu, and J. Y. Chung, "Algorithms for Scheduling Imprecise Computations with Timing Constraints," *SIAM J. Comput.*, vol. 20, no. 3, pp. 537–552, June 1991.

[79] K. G. Shin, C. M. Krishna, and Y. H. Lee, "A Unified Method for Evaluating Real-Time Computer Controllers and Its Applications," *Trans. on Automat. Contr.*, vol. AC-38, no. 4, pp. 357–366, IEEE, Apr. 1985.

[80] D. Sleator, "A 2.5 Times Optimal Algorithm for Packing in Two Dimensions," *Information Processing Letters*, vol. 10, no. 1, pp. 37–40, Feb. 1980.

[81] K. P. Smith and J. W. S. Liu, "Monotonically Improving Approximate Answers to Relational Algebra Queries," *Proc. of Computer Software and Applications Conf.*, 1989.

[82] B. Sprunt, J. Lehoczky, and L. Sha, "Exploiting Unused Periodic Time for Aperiodic Service Using the Extended Priority Exchange Algorithm," *Proc. 9th Real-Time Systems Symp.*, pp. 251–258, IEEE, Dec. 1988.

[83] J. A. Stankovic, K. Ramamritham, and S. Cheng, "Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-time Systems," *Trans. on Computers*, vol. C-34, pp. 1130–1143, IEEE, Dec. 1985.

[84] J. A. Stankovic, "Misconceptions about Real-Time Computing," *Computer*, pp. 10–19, IEEE, Oct. 1988.

[85] A. Stoyenko, "A Schedulability Analyzer for Real-Time Euclid," *Proc. 8th Real-Time Systems Symp.*, pp. 218–227, IEEE, Dec. 1987.

[86] R. Strong, D. Dolev, and F. Cristian, "New Latency Bounds for Atomic Broadcasts," *Proc. 11th Real-Time Systems Symp.*, pp. 156–165, IEEE, Dec. 1990.

[87] R. E. Tarjan, *Data Structures and Network Algorithms,* SIAM, Philadelphia, PA, 1983.

[88] P. Tu and K. Lin, "Scheduling Performance Polymorphic Computations in Real–Time Systems," *Proc. 15th Int'l Computer Software and Applications Conf.*, pp. 406–411, Sep. 1991.

[89] P. Vaidya, "An Algorithm for Linear Programming which Requires $O(((m+n)n^2+(m+n)^{1.5}n)L)$ Arithmetic Operations," *Math. Prog.*, no. 47, pp. 175–201, 1990.

[90] S. Vrbsky and J. W. S. Liu, "An Object–Oriented Query Processor That Returns Monotonically Improving Answers," *Proc. Int'l Conf. on Data Eng.*, pp. 472–481, IEEE, Apr. 1991.

[91] B. W. Wah, G. J. Li, and C. F. Yu, "Multiprocessing of Combinatorial Search Problems," *IEEE Computer*, vol. 18, no. 6, pp. 93–108, Also in *Tutorial: Computers for Artificial Intelligence Applications*, ed. B. W. Wah, IEEE Computer Society, 1986, pp. 173–188., June 1985.

[92] J. Wakely, *Error Detecting Codes, Self–Checking Circuits and Applications,* North–Holland, New York, NY, 1982.

[93] G. Wallace, "Overview of the JPEG (ISO/CCITT) Still Image Compression Standard," *Visual Communication and Image Processing, '89, SPIE*, Nov. 1989.

[94] J. Wensley, L. Lamport, J. Goldberg, M. Green, K. Levitt, P. Melliar–Smith, R. Shostak, and C. Weinstock, "SIFT: Design and Analysis of a Fault–Tolerant Computer for Aircraft Control," *Proc. of the IEEE*, vol. 66, no. 10, pp. 1240–1255, Oct. 1978.

[95] Y. Wu and T. Lewis, "Parallel Algorithms for Decomposible Linear Programs," *Proc. Int'l Conf. on Parallel Processing*, vol. 3, pp. 27–34, IEEE, 1990.

[96] W. A. Wulf and C. G. Bell, "C.mmp—A Multi–mini Processor," *Proc. Fall Joint Comp. Conf.*, vol. 41, pp. 765–777, AFIPS Press, 1972.

[97] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real–Time Systems," *IEEE Trans. Soft. Eng.*, vol. SE–13, no. 5, pp. 564–577, Aug. 1987.

[98] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Preemptive Scheduling under Time and Resource Constraints," *Trans. on Computers*, vol. C–36, no. 8, pp. 949–961, IEEE, Aug. 1987.

[99] W. Zhao and E. Chong, "Performance Evaluation of Scheduling Algorithms for Dynamic Imprecise Soft Real–Time Computer Systems," *Australian Computer Science Communications*, vol. 11, no. 1, pp. 329–340, 1989.

Vita

Albert Chuang–Shi Yu was born on [redacted] in [redacted] He attended the University of California, Berkeley, where he received a double B.A. degree in Computer Science and Physics in 1985. His M.S. and Ph.D. degrees in Computer Science from the University of Illinois at Urbana–Champaign were conferred in Jan. 1990 and May 1992 respectively.

Albert was a software engineer of VLSI CAD tools in the design automation group at Memorex Corp. (now, Unisys) from 1984 to 1985. He joined the communication network group at IBM T. J. Watson Research Center in Summer 1986. He became a research assistant with Prof. Benjamin Wah in the Center for Reliable and High–Performance Computing (formally, the Computer System Group) in Fall 1986. He spent three summers from 1987 to 1989 in the Intelligent Systems Technology Branch at the NASA Ames Research Center where he conducted research on multiprocessing of Lisp programs. He was awarded a NASA Fellowship (Graduate Student Researchers Program) from 1988 to 1991 and a Southland Illini Club of Los Angeles Scholarship in 1989. From 1990 to 1992, he performed research on real–time systems under the supervision of Prof. Kwei–Jay Lin. He is currently a member of the technical staff at the Hughes Aircraft Company.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-92-1738	2.	3. Recipient's Accession No.

4. Title and Subtitle

SCHEDULING PARALLEL REAL-TIME TASKS THAT ALLOW IMPRECISE RESULTS

5. Report Date March 1992

6.

7. Author(s)

Albert C. Yu

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

University of Illinois at Urbana-Champaign
Department of Computer Science
1304 West Springfield Avenue
Urbana, IL 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

NASA NGT-50349
NASA NGT-1-613
N0014-89-J-1181

12. Sponsoring Organization Name and Address

National Aeronautics and Space Administration
Office of Naval Research
Washington, DC

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

Imprecise computation and parallel processing are two techniques for avoiding timing faults and tolerating hardware faults in hard real-time systems. When a result of the desired quality cannot be produced in time, hard real-time systems can produce an intermediate result of acceptable quality by imprecise computation, reduce the response time of the result by parallel processing, or both, to avoid timing faults. To mask hardware faults, a real-time task is replicated into several copies which are executed on distinct processing elements. This thesis describes efficient algorithms for scheduling two different task models on multiprocessors. Both task models support the imprecise computation technique whereby each task is logically decomposed into a hard task and a soft task. The hard task must be completed before the deadline to produce an acceptable result. The soft task refines the result produced by the hard task until the deadline. In the parallelizable task model, each task may be decomposed into concurrent subtasks which are processed simultaneously by multiple processing elements. The overhead associated with concurrent processing is assumed to be a linear function of the degree of parallelism. The scheduling algorithm for this model is optimal, if the multiprocessing overhead is indeed a linear function of the degree of parallelism. In the replicated task model, the replicas of a task must be assigned to distinct processing elements. The performance of the scheduling algorithms for this model is evaluated by stochastic analysis and computer simulations.

17. Key Words and Document Analysis. 17a. Descriptors

Keywords: real-time systems, imprecise computation, parallel processing, multiprocessor scheduling, fault-tolerant computing, task reconfiguration, linear programming.

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 136
	20. Security Class (This Page) UNCLASSIFIED	22. Price